In 29th Annual Hawaii International Conference on System Sciences,
H. El-Rewini and B. D. Shriver, (eds), IEEE Computer Society Press, 741-750, 1996

A Formal Framework For Specifying Design Methods

Mark d'Inverno, G. R. Ribeiro Justo, Paul Howells
School of Computer Science
University of Westminster, London
e-mail: {dinverm,justog,howellp}@wmin.ac.uk

Abstract

The main objective of this paper is to put forward a software process model for high-performance systems (HPS), and to present a formal framework to describe software design methodologies (SDMs) for those systems. The framework consists of two main parts: the software process activities which characterise the development of HPS, and the components of the SDM (concepts, artifacts, representation and actions) which are essential for any methodology. The framework relates these two parts by identifying generic components of each activity in the software process that can be used to classify and evaluate SDMs for HPS. The framework has been formally specified using the language Z and used to derive formal specifications of SDMs. This is illustrated in the paper by presenting part of the specification of oDM *(an occam design method).*

1 Introduction

The number of software design methodologies (SDMs) currently available is enormous. This means that designers and managers are faced with the difficult task of deciding which SDM best suits their projects. Methodology and tool developers also face problems as their methodologies and tools must evolve to cope with technological changes.

In order to solve these problems, there is an urgent need for models or frameworks which allow us to characterise, classify and integrate SDMs and many authors have proposed such solutions [5, 24, 3, 2]. However, most of them concentrate only on particular properties of the life-cycle model or software process model on which SDMs are based, and whilst still important in enabling users to evaluate properties of particular SDMs [5], do not go further in considering the structure of the SDMs and their components. When solutions do consider their structure and components [23] it becomes possible not only to give a more detailed comparison and evaluation of systems, but to allow for the integration of different components of SDMs.

Unfortunately, most of these models and frameworks are based on traditional (sequential) SDMs. Little has been done to systematically classify and evaluate SDMs and tools for high-performance systems (HPS) the field where parallel computing is applied. Attempts have been made in presenting surveys of methods and tools applied to the development of HPS [14] but there are not frameworks or models similar to those applied to traditional software. There are many reasons for this, the main one being the lack of life-cycle or software process models that describe the development of HPS. The principal consequence is that HPS developers cannot systematically compare the few SDMs available. In addition, tool developers have little insight into how their tools can be applied and integrated to SDMs.

The main objective of this paper is to propose a characterisation of the HPS software process, and present a formal framework to describe SDMs for HPS. Although the framework aims at the systematic classification of methods, it can also be applied to the classification of tools by identifying which components or activities they can support.

In the following section, a model of the software process for HPS is introduced. This model is the basis for the framework which is presented in Section 3. First, we describe existing frameworks and then informally introduce our own. The outline of the formal specification in Z of the framework is presented in Section 4. Section 5 presents an example of how the framework has been used to specify an existing SDM for HPS. Finally, Section 6 presents the conclusions of the paper, and future work.

2 The HP Software Process

Independently of the approach taken (life cycle models [21, 19] or software process models[2]), traditional software development has been fully characterised by its activities and their relationships. Unfortunately, little work has been done to characterise the HPS software process, one of the reasons being the young state of the field. Another important reason is the historical development of HPS themselves; much of the initial work on the development of HPS consisted of the transformation of legacy (sequential) code into parallel code. This activity was supposed to be performed automatically by parallelising compilers [22]. However, the resulting parallelism was considered harmful and had to be hidden from the user, and consequently HPS development was viewed merely as a small variation of the traditional software process. The limitations of this approach have now been recognised [1] and it is clear that the programmer/designer must share the responsibility of exposing parallelism. In fact, the advantages of explicit parallelism have been recognised by certain authors for some time [10], and there is now an urgent need for more SDMs (and tools) which support the development of explicitly parallel programs [14].

Our process model is described in statechart [9] (Figure 1); a visual formalism which extends the notion of state diagram with hierarchy and abstraction. States (seen as sets) are denoted by bullets and arrows represent transitions. If two sub-states (sets) B and C of state A do not overlap, they are exclusive. In this case, state A can be in sub-state B or C. Default arrows indicate default initial states. A

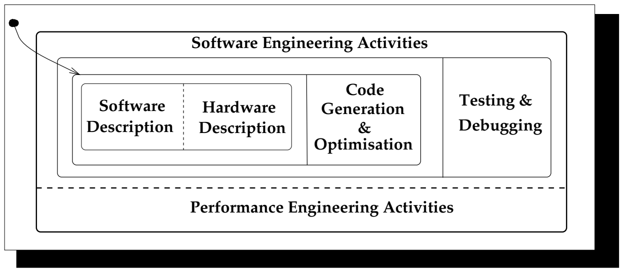
Software Engineering Activities
Software Description
Hardware Description
Code Generation & Optimisation
Testing & Debugging
Performance Engineering Activities

Figure 1: A model of a HP software process.

default arrow pointing to a non-atomic state indicates that every sub-state is a possible initial state. Dotted lines stands for the (unordered) *Cartesian product* of states.

One of the main differences between traditional software development and HPS development is that in the former, performance usually plays a minor role and is only considered in the final stages of the development. For HPS, however, performance is a dominant issue and some authors claim that it should be considered in all stages of the development [8, 15]. This means that there is some tension between the performance activities (performance engineering activities) and the "traditional" development activities (software engineering activities). As these activities are usually related in different ways by tools and methods, we have decided to consider a concurrent view of these groups of activities (as illustrated in Figure 1). This means, for example, that performance prototypes can be developed both in the early stages of the development [8, 15], and after the implementation [4].

We now describe each group of activities, starting from the software engineering activities which are the (default) initial state. In most cases the development starts with the problem specification, and the eventual *description* of a solution design (software). Traditionally, an implementation (code) is derived later from this description, possibly assuming a certain environment (target machine). In terms of HPS, however, information about the *target machine* (hardware) will not only influence the implementation but it may influence the solution itself [8]. So, the model allows for the *hardware description* to be constructed and analysed simultaneously with the software. This enables us to model SDMs where the hardware-dependent designs are derived from abstract (virtual) designs [13], and also those where the hardware characteristics are taken into account during the design, especially to reduce "performance design errors" [4].

Certainly, an implementation is only derived after a software/hardware description has been produced. However, many approaches assume the transformation of existing programs known as parallelisation. Our model accommodates these approaches by allowing the software engineering activities to start from the code generation state. Finally, the correctness of the implementation can be tested, and possible errors can be traced by using debugging tools.

Performance engineering activities consist of measuring the performance and then analysing of the results. For simplification, the description of our model presented in this paper shows the performance engineering activities as a single activity. The performance measurements (statistics) can be obtained before the program has been implemented (using performance prediction methods) or after (using prediction or monitoring methods). The statistics about program performance produced via prediction or monitoring are then analysed, typically with the support of visualisation tools. The results can then be used to modify the program or the design to improve performance. It is important to understand that the performance measurements do not simply include time or resource usage but also *scalability* and *portability*, which measure the relationship between the software and hardware.

3 The Framework

3.1 Some Existing Frameworks

Davis, and Comer [5] present a framework to compare life cycle models. This is based on five metrics which measure properties of the life cycle and how it responds to the evolution of the users' needs. This framework can help developers in evaluating how their methodologies may react to the user needs but as it concentrates on major properties, it does not help in the integration of components of life cycle models or in the evaluation of supporting tools.

The framework proposed by Conrandi, Ferbström and Fuggetta [3] concentrates on the evolutionary properties of the software development process: it is particularly important for the systematic evaluation of the capabilities of SDMs to support evolution.

The framework proposed by Blum [2] focuses not on life cycle models but on the software process. The author defines the software process as a series of activities – with respect to a software product – from conception to implementation. He also identifies two major domains in the software process: the problem domain where the problem is solved, and the implementation domain where a solution to the problem is executed. So, problem-oriented SDMs concentrate on producing a better understanding of the problem and its solution and product-oriented SDMs concentrate on the generation of an implementation from a specification. Another dimension of the framework is the separation of SDMs based on conceptual models from those based on formal models. Conceptual models describe (subjective) guidelines for design decisions and validation, whilst formal models prescribe (objective) criteria for correct behaviour of the software product.

The above frameworks focus on general properties of SDMs, but not on their structure and components. They do not provide the support for integration of SDMs, for example. By contrast, the framework presented by Song and Osterweil [24] describes the components of SDMs based on the empirical observation of a group of existing and relevant SDMs. The components are organised in a hierarchical way where the top level assumes that the objective of SDMs is to provide a systematic way (actions) for the production of artifacts using different representations, following some principles and guidelines (concepts). These elements are then further decomposed. Thus the framework allows a detailed description of SDMs and therefore can also be used to define the possible integration of components of different methods.

3.2 Informal Description

Our framework follows the same principle of that presented in [24] where the basic components of SDMs are identified, and then categorised. These components are also: **Concepts**, **Artifacts**, **Representation** and **Actions**. However, our view of what these elements are differs from that suggested by Song and Osterweil [24]. One of the main reasons is that we want to present a framework which is formally specified, and as such, some abstract notions considered by Song and Osterweil are not relevant. Another important point is the nature of these components, which depends very much on the SDMs to which the framework will be applied. Song and Osterweil are interested in problem-oriented methods whilst we are interested in solution-oriented methods; the kind of methods usually available for HPS.

In summary, SDMs, for us, consist of **Concepts** which are basic elements that are related to form **Artifacts** which are produced and manipulated by **Actions**. These artifacts can be realised using different concrete **Representations**. An informal description of each component is presented below. In the next section they will be specified formally.

Concepts denote the basic (usually atomic) elements that are used in a method; they are the raw product of the **Artifacts**. They can be, for example, a *task* in [20] or a *template* in [16]. The framework presents **Concepts** which are general enough to characterise many SDMs.

Artifacts are structured elements defined from **Concepts** or other **Artifacts**. They usually define relationships amongst **Concepts** and **Artifacts**. For example, a *configuration* of *task*s in [20] or a *network* of *templates* in [16] define a relationship amongst the *task*s or *template*s respectively.

Representation is a concrete realisation of **Artifacts**. The type of **Representation** of the **Artifacts** defines the nature of the method in terms of its formality and visual basis. The framework defines four categories of **Representation**: *Textual*, *Textual_Formal*, *Structural* and *Formal_Visual*. The first category *Textual*, denotes pieces of text in an informal language which has no formal semantics. The second category is also textual but is defined in a formal language, for example, CSP [10]. The last two categories comprise graphical representations but the *Structural* ones do not have a formal semantics and are simply manipulated as pictures whilst the *Formal_Visual* have a formal semantics and can be formally manipulated. For example, a *configuration diagram* in [20] is classified as *Structural* but a Petri Net in [8] is *Formal_Visual*.

Actions denote the operations provided by SDMs for the production and manipulation of the **Artifacts**. The framework provides a collection of operations which we believe are sufficient to characterise most SDMs for HPS.

Having identified the basic components of the SDM, the next part of the framework consists of a collection of basic components for the software process model, which was presented in Section 2. This means that for each activity of the software process its basic components (**Concepts**, **Artifacts**, **Representation** and **Actions**) are identified. The complete structure of the framework is presented in Table 1. Below we describe the components of each activity:

Software Description: the main issue of this activity is to decompose the system into a collection of parallel units and to describe the relationship between them [20]. In this case, the framework provides the concept of *Parallelism Units* which are structured into the *Design_Structure* that relates the *Parallelism Unit*s. Another important aspect of the software description is the way the *Parallelism Unit*s behave or interact among themselves. In the framework, this is defined by the *Behaviour_Description*. The *Design_Structure* is usually represented in a *Structural* way. Most SDMs for HPS include some form of graphical representation [14]. However, these representations do not present a well-defined semantics and are only used to illustrate the *Design_Structure*. The *Behaviour_Description* is usually represented in a *Textual* form like a piece of code or more formally (*Formal_Textual*) using a specification language [14]. There are, however, cases where *Formal_visual* representations are used [14].

These artifacts are built using actions for *Decompose* and *Identify Parallelism Unit*s or *Selecting* pre-defined artifacts. The *Behaviour_Description* can also be specified (*Specify*) or built from pre-defined descriptions [16]. The *Mapping_Description* relates artifacts from the *Software* and *Hardware Description*.

Hardware Description: this activity is based on the concepts of hardware components which are essential for a (parallel) machine model [26]. This description is usually presented in a simple declarative way. *Textual* representations indicating the hardware parameters are the most common but graphical representations are also used [27].

Code Generation & Optimisation: this activity is concerned with the concept of *Program_Structure*s which are put together to form the *Program_Code*. These structures can denote loops [1] or other elementary program components [16]. The process of code generation and optimisation basically consist of *Derive_Code* from the *Software Description* artifacts and *Transform_Code* to make it more suitable for the specific hardware.

Testing & Debugging: the concepts of *Test* and *Error* are the basic elements of this Activity. These elements are usually structured as lists which are mostly represented textually but sometimes graphically [14].

Performance_Engineering: this activity consists of the derivation of *Performance_Statistics* about the *Program_Code* or *Design_Structure* [8]. Again, these artifacts are represented textually but there is some mechanism to represent them graphically [15].

One of the important aspects of the framework, as we will see in the next section, is to enable the SDM developers, and also their users, to classify the components of a SDM in a systematic way. Table 1 corresponds to an archetype methodology and the intention is to identify which of those components a particular methodology contains. The next section presents a formal description of the framework in terms of Z.

Software Process Activities	**Framework Components**			
	Concepts	Artifacts	Representation	Actions
Software Description	*Parallelism Unit* *Mode of Parallelism* *Synchronisation Mode*	*Design_Structure (DS)* *Message_Structure (MS)* *Behaviour_Description* *Mapping_Description*	*Structural* *Textual* *Formal_Textual* *Formal_Visual*	*Decompose_PU* *Specify_MS* *Specify_Interactions* *Specify_Mapping* *Specify_PU_Behaviour* *Verify_Interactions* *Derive_DS* *Refine_DS* *Select_Behaviour_Template* *Select_Mapping_Template* *Select_Message_Template* *Modify_Struct/Descr*
Hardware Description	*Processor Type* *Memory Type* *Network Type*	*Processor_Description* *Memory_Description* *Network_Description* *Virtual_Description*	*Structural* *Formal_Textual* *Textual*	*Specify_Processor* *Specify_Memory* *Specify_Network* *Select_Network_Template* *Specify_Virtual_Network* *Verify_SH_Suitability* *Modify_Description*
Code Generation & Optimisation	*Program_Structure* *Optimisation*	*Program_Code* *(sequential & parallel)* *Program_Skeleton*	*Formal_Textual* *Structural* *Formal_Visual*	*Derive_Code* *Transform_Code* *Select_Transformation* *Apply_Transformation*
Testing & Debugging	*Correctness* *Test* *Error* *Program*	*List_of_Tests* *List_of_Errors* *Program_Code* *Program_Skeleton*	*Formal_Textual* *Structural* *Formal_Visual*	*Identify_Error* *Specify_Test* *Apply_Test* *Check_Property*
Performance Engineering	*Performance evaluation* *Portability* *Suitability*	*Performance_Model* *Performance_Statistics* *List_of_Events*	*Formal_Textual* *Structural* *Formal_Visual*	*Select_Statistics* *Select_Events* *Identify_Bottlenecks* *Check_Property* *Specify_Model* *Derive_Model* *Derive_Statistics* *Derive_Events*

Table 1: Classification of the components of the framework using the process model.

4 Formalising the Framework

4.1 Introduction to Z

The Z language [25] is based on set theory and first order logic. It extends the use of these languages by allowing an additional mathematical type known as the schema type. Syntactically a schema is a box divided into two parts by a horizontal line. There are two types of schema: state and peration. In a state schema, the upper half is known as the declarative part, and is used to declare variables and their types. The second part of the state schema is known as the predicate part, and in this part we show how the variables are related and constrained. Each schema has a distinct name. Semantically, a schema can be considered as having the same type as the *Cartesian* product of the types of its variables, without ordering, and with the state space constrained by the schema predicates. State schemas describe the possible states of a system. Modularity is facilitated in Z by allowing schemas to be included within other schemas.

Operations effect state, and are characterised by their effect on the state. An operation schema relates the state variables before and after the operation. The general operation schema has a before state (unprimed state variables), an after state (primed state variables), inputs (question-marked variables), outputs (exclamation-marked variables), and a set of pre-conditions for the application of the operation.

To introduce a type where the structure of the elements is not considered, we use the notion of a given set. For example, $[EVENT]$ represents the set of all events, and we write $events : \mathbb{P}\, EVENT$ to declare a set of events. A relation ($\leftrightarrow$), expresses some relationship between two existing types known as the source and target of the relation. Total functions ($\rightarrow$) and partial ($\nrightarrow$) can also be defined. In addition, Z provides a schema calculus which allows schemas to be joined using the basic logical connectives such as disjunction ($\vee$). The use of Z in providing frameworks for the presentation, evaluation and comparison of classes of systems has been shown elsewhere [6, 7, 17, 18] and will not be considered further here.

4.2 Specifying the Framework in Z

We now provide an overview of how the framework represents the concepts, artifacts, representations and actions of a methodology, and then describe the $structure$ of the framework specification. Initially, we define the concepts of the system as higher-level given sets. It may also be necessary to define given sets to represent certain secondary concepts which can not be expressed in terms of the primary concepts.

Artifacts are the elements which are defined from the concepts and are represented mathematically as some collection of state variables – which we call artifact_variables – each describing some aspect of an artifact. The type of any artifact_variable can (and must) be expressed solely in terms of the basic concept types. For example, suppose that we wish to constrain particular sequences of event along a certain channel. One means of representing this would be to define some artifact_variable with the following type

$$artifact\_var1 : Channel \times \mathbb{P}(\mathrm{seq}\, Event)$$

so that $artifact\_var1$ could represent the set of possible sequences of events allowed on a particular channel. This would be legitimate so long as both $Event$ and $Channel$ are concepts of the methodology. Artifact_variables thus represent the relationships under investigation between concepts, between concepts and artifacts and so on hierarchically, building up the necessary information structures to describe the artifacts. Initially before any actions of the method have been performed these variables are set either to the empty set or to some system global constant. Further, within the framework we can define state schema predicates which ensure that the artifact_variables are always well-defined.

It is through repeated use of the actions of a method that the designer can develop, decompose and refine the artifacts of the system. The definition of these actions is done in terms of operation schemas which effect certain artifact_variables. Lastly, representation of the artifacts can take several forms such as code fragments, textual descriptions and structured diagrams.

Within our framework we have defined a set of generic state and operation schemas representing the possible artifacts and actions given in Table 1. We then draw from this library as required.

4.2.1 The Specification Structure

The basic premise is to separate the concerns of artifacts into distinct but related part_activities. A part_activity corresponds to some aspect or sub-method of the whole activity which can be considered independently. Each part_activity is then concerned with a particular artifact and set of artifact_variables, which are created only through the use of the $Actions$ associated with that part_activity. It might be that some part_activities have to be ordered since artifact_variables of one part_activity may rely on existing $artifact\_variabes$ being defined from another part_activity. Further than this, formalising an activity in this way enables the designer to understand the nature of information dependency within a design methodology.

Accordingly, the approach taken in the specification is to describe each of the part_activities separately. For each, we define the general state, the initial state, the terminal state and the operations/actions associated with that part_activity which allow the designer to achieve the terminal state from the initial state. These terminal states can then be used by automatic system operations known as system-inferences which can then derive both additional design information from the terminal states and representations of the design information as required. For completeness in the framework the automatic inference activities are included in the general state of succeeding part_activities.

Each action is specified by one or more schemas describing the *successful* completion of the operation and one operation schema for each possible *error* case of the action. Each error schema specifies a report to the user indicating why the action could not be performed. The *total* operation is then constructed from these schemas.

However, it may be that for any given part_activity is large and itself contains separate concerns which are so related that they must be treated simultaneously rather than sequentially. (In fact, it may be that a method does not lend itself to be considered in sequential part_activities at all and so must be be considered in its entirety). The framework thus allows for each part_activity's state to be split into sub-states, each described by a separate sub-state schema which records the information introduced by the designer during that part_activity. This is another direction along which modularisation can take place within the framework and allows large part_activities to be decomposed into manageable sub-states where the relationship between the sub-part_activities is well-defined. The initial and terminal states for each part_activity are then composed of the initial states and terminal states respectively, of each of the part_activity's sub-states.

4.3 The Components of the Framework

For each part_activity of the method we define:

1. The sub-states: as stated before, the framework makes use of generic state and operation schemas. These can then be re-used as required in describing the relations that exist in a method and the actions which are defined on these relations. In the following simple example we introduce a generic schema which states that every (generic) *element* has an associated *textual description*.

   ```
   ── Element–State[X] ──────────────────
   │ knownElements : 𝔽 X
   │ ElementDesc : X ⤔ DESC
   ├──────────────────────────────
   │ dom ElementDesc ⊆ knownElements
   └──────────────────────────────────
   ```

   As we will see, this schema may then be re-used with the generic instantiated to the relevant artifact_variable under consideration.

2. The general state: using schema inclusion we define the general state as the collection of sub-states together with any additional constraints between the artifacts of the sub-states included in the predicate part of the schema.

   ```
   ── StageNState ──────────────────────
   │ PartActivityN–Sub-State₁
   │ ⋮
   │ PartActivityN–Sub-Stateₘ
   ├──────────────────────────────
   │ Predicates_Relating_Sub-States
   └──────────────────────────────────
   ```

3. The initial sub-states: before any action the variables are undefined or set to system global-variables. For example, the initial state of the previous generic schema is given by:

   ```
   ── Init–Element–State[X] ─────────────
   │ Element–State[X]
   ├──────────────────────────────
   │ knownElements = ∅
   │ ElementDesc = ∅
   └──────────────────────────────────
   ```

4. The general initial state: if there are m sub-states for $PartActivityN$, then the general initial state is written:

   ```
   ── InitStageNState ──────────────────────
   │ Initial–PartActivityN–Sub-State_1
   │ ⋮
   │ Initial–PartActivityN–Sub-State_m
   └────────────────────────────────────────
   ```

5. Actions: all operations which effect a sub-state are defined with pre-conditions ensuring that information is consistently maintained. For example, consider the $identify$ action represented by the following operation schema which creates a new element from some type and provides a textual description of this element so updating the state of the *Element–State[X]* schema previously given.

   ```
   ── Identify–Element[X] ──────────────────
   │ Δ Element–State[X]
   │ element? : X
   │ description? : DESC
   ├──────────────────────────────────
   │ element? ∉ knownElements
   │ knownElements′ = knownElements ∪ {element?}
   │ ElementDesc′ = ElementDesc ∪
   │                  {(element?, description?)}
   └────────────────────────────────────────
   ```

   Operations are defined on the entire part_activity but in many cases only one of the sub-states of a given part_activity is actually altered. In response, we introduce a new schema convention $\oslash$*PartActivityN–Sub-State*$_i$ to be used to permit changes to only one of the component sub-state schemas of the composite state schema which represents a given part_activity. This not only makes the schemas much more concise, but much more readable since the sub-state that is being altered is explicitly mentioned.

   ```
   ── ⊘PartActivityN–Sub-State_i ───────────
   │ Δ StageNState
   │
   │ Ξ PartActivityN–Sub-State_1
   │ ⋮
   │ Ξ PartActivityN–Sub-State_{i−1}
   │
   │ Δ PartActivityN–Sub-State_i
   │
   │ Ξ PartActivityN–Sub-State_{i+1}
   │ ⋮
   │ Ξ PartActivityN–Sub-State_m
   └────────────────────────────────────────
   ```

6. Terminal states: the terminal sub-states are then defined. For example, the *terminal* generic state schema for *Element–State[X]* is given as follows. It insists that every identified element has an associated description. The general terminal state can be defined in terms of the terminal sub-states.

   ```
   ── Term–Element–State[X] ────────────────
   │ Element–State[X]
   ├──────────────────────────────────
   │ dom ElementDesc = knownElements
   └────────────────────────────────────────
   ```

5 Application of the Framework to oDM

The occam design method (oDM) is currently targeted at the production of occam programs [12]. It is based on the use of Data Flow Modelling (DFM) together with aspects of CSP [10] and FOREST [11]. The motivation for adopting a DFM approach is that data flow diagrams (DFDs) provide a good indication of the structure of a program and map directly into occam. The Data Flow Model is a hierarchical collection of data flow diagrams. The top-level DFD represents the main program, where nodes represent processes (possibly executing in parallel) and arcs represent communication channels which connect these processes. Each process itself may be decomposed recursively into sub-processes, which themselves are represented by DFDs.

oDM comprises eight stages which are suitable only for the top-level design, together with two stages which are only appropriate to the internal (lower-level) design of processes. The strategy for applying oDM is to recursively perform the stages until the designer is left only with sequential processes. The top-level design stages are: Process and Action Structure, Data Analysis, Data-flow Analysis, Protocol Analysis, Communication Analysis, Processes Termination Analysis, Procedure Specification and Main Body Construction. The next phase is to repeat these stages for each of the identified top-level processes. As previously stated, processes are recursively decomposed into subprocesses, each of which is represented by a new DFD. The lower-level design stages include all top-level design stages plus the following: Process Structure Analysis and Message Sending/Receiving Analysis.

Each stage of oDM is concerned with one aspect of program design. The general approach is that, by completing the stage, a program designer can add to the Data Flow Model and so incrementally develop the design.

As Table 2 illustrates, according to our framework oDM is concerned with the software description activities. Then, using the framework, we can identify the following actions of oDM:

- *Identify* creates an instance of a concept-type. In oDM this is always associated with supplying with this new instance a textual description of its role. (The generic state and operation schemas relating to this action were given in the previous section).
- *Define* creates a relationship between an artifact and another (set of) artifact(s).
- *Create* chooses an instance of a particular relationship which inherits the properties of that relationship. (For example in oDM if a process is created of a certain process-class then the process contains an action of each of the actionclasses associated with that process-class.)
- *Modify_Add* takes a definition and creates an extra mapping between an artifact and a set of artifacts.
- *Modify_Delete* takes a definition and removes one existing relationship between an artifact and set of artifacts.
- *Select* takes an artifact and places it in a particular category of concept.

Software Process Activities	**Framework Components**			
	Concepts	Artifacts	Representation	Events
Software Description	*Process* *Event* *Channel*	*Design_Structure* *Message_Structure* *Behaviour_Description*	*Formal_Textual (eg Occam Code)* *Formal_Visual (eg DFDs)* *Textual (eg concept descriptions)*	*Identify (describe)* *Define* *Create* *Modify_add* *Modify_Delete* *Verify* *Select* *Derive*

Table 2: Classification of the components of oDM using the framework.

- *Verify* takes some collection of artifacts and checks the consistency of the information.
- *Derive* takes information contained in existing artifacts and generates either new artifacts or representations.

5.1 Specification of oDM

We now show how oDM has been specified within the formal framework. Firstly, we defined the primary concepts of oDM as given in Table 2 using given sets:

$$[\,PROCESS,\ EVENT,\ CHANNEL]$$

The secondary concepts are given by

$$[\,DESC\,]$$

Further, for the purposes of reporting, we build up new types from these concept-types as follows:

$$\begin{array}{ll}
OP & ::= \mathtt{Create} \mid \mathtt{Define} \mid \mathtt{Identify} \mid \ldots \\
OBJ & ::= \mathtt{Event} \mid \mathtt{Process} \mid \mathtt{Channel} \mid \mathtt{Desc} \\
COND & ::= \mathtt{Already\text{-}Created} \mid \mathtt{Already\text{-}Defined} \mid \ldots \\
REP & ::= \ \mathsf{succ}\langle\!\langle OP \times OBJ \rangle\!\rangle \mid \mathsf{err}\langle\!\langle OP \times OBJ \times COND \rangle\!\rangle
\end{array}$$

We then treat each of the ten stages of oDM as part_activities. For the purposes of illustration we will show how the first part_activity called part_activity1 is derived formally from the framework. This part_activity is concerned with identifying the processes and events of a program and defining a higher level Behaviour_Description in terms of the set of events which a process can perform. Table 3 illustrates part_activity1 of oDM.

Before we start the description of part_activity1 we introduce the following type synonyms, defined in terms of our basic concepts. An EventClass is a set of Events and a ProcessClass is a set of Processes.

$$\begin{array}{l}
EVCLASS == \mathbb{P}\, EVENT \\
PROCCLASS == \mathbb{P}\, PROCESS
\end{array}$$

We now show how the specification of the oDM part_activity1 can be given in terms of the framework components given in section 4.3.

1. The sub-states: part_activity1's state is composed of the four sub-states as follows.

   EventClassState: EventClasses identified by the designer and *EventClassDesc* is a mapping from the EventClasses to their natural language descriptions. Only known EventClasses can be described. This state schema is then an instantiation of the generic library schema *Element–State[X]* with variables re-named as follows.

$$\begin{array}{l}
EventClassState \mathrel{\widehat=} Element\text{–}State[EVCLASS] \\
\qquad [\,knownEventClasses / knownElements, \\
\qquad\qquad EventClassDesc / ElementDesc\,]
\end{array}$$

Which is equivalent to:

$$\begin{array}{|l}
\hline EventClassState \\
knownEventClasses : \mathbb{F}\, EVCLASS \\
EventClassDesc : EVCLASS \rightarrowtail\!\!\!\!\!\!\rightarrow DESC \\
\hline
\mathrm{dom}\ EventClassDesc \subseteq knownEventClasses \\
\hline
\end{array}$$

The following sub-state schemas are similarly obtained through library schemas.

ProcClassState: here, *AlphabetClass* is a mapping from the ProcessClasses to a set of EventClasses, *knownProcClasses* is the set of all the ProcessClasses defined by the designer and *ProcClassDesc* is a mapping from ProcessClasses to their descriptions. Further, every known ProcessClass must have a set of EventClasses and only known ProcessClasses can be described.

$$\begin{array}{|l}
\hline ProcClassState \\
AlphabetClass : PROCCLASS \twoheadrightarrow\!\!\!\!\!\!\!\!+\ \ \mathbb{F}\, EVCLASS \\
knownProcClasses : \mathbb{F}\, PROCCLASS \\
ProcClassDesc : PROCCLASS \rightarrowtail\!\!\!\!\!\!\rightarrow DESC \\
\hline
knownProcClasses = \mathrm{dom}\ AlphabetClass \\
\mathrm{dom}\ ProcClassDesc \subseteq knownProcClasses \\
\hline
\end{array}$$

ResClassState: we have that *knownResClasses* is the set of all system Resource Classes.

$$\begin{array}{|l}
\hline ResClassState \\
knownResClasses : \mathbb{F}\, PROCCLASS \\
\hline
\end{array}$$

ProcState: in this schema we have that *classofProcess* and *classofEvent* are mappings from Processes and Events to their ProcessClass and EventClass respectively, *knownProcesses* is the set of all the defined Processes, *knownActions* is the set of all the Events created by the system and *alphabet* is a mapping

Software Process Activities	**Framework Components**			
	Concepts	Artifacts	Representation	Events

Software Description (part_activity1)	*Process*
	Action
		Design_Structure	*Textual*
			Formal_Visual
				Identify (ActionClass)
				Define(ProcessClass)
				Create(Process)
				Derive(Events)

Table 3: Classification of the components of part_activity1 using the framework.

from a Process to its Event alphabet. In addition, every known event/process belongs to an EventClass/ProcessClass, only known processes can have their event alphabets defined, and only known Events can be used in the alphabet of a Process.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ProcState}} \\ \hline
\mathit{classofProcess} : \mathit{PROCESS} \pfun \mathit{PROCCLASS} \\
\mathit{classofEvent} : \mathit{EVENT} \pfun \mathit{EVCLASS} \\
\mathit{knownProcesses} : \mathbb{F}\ \mathit{PROCESS} \\
\mathit{knownActions} : \mathbb{F}\ \mathit{EVENT} \\
\mathit{alphabet} : \mathit{PROCESS} \pfun \mathbb{F}\ \mathit{EVENT} \\ \hline
\mathit{knownProcesses} = \mathrm{dom}\ \mathit{classofProcess} \\
\mathit{knownActions} = \mathrm{dom}\ \mathit{classofEvent} \\
\\
\mathrm{dom}\ \mathit{alphabet} \subseteq \mathit{knownProcesses} \\
\bigcup(\mathrm{ran}\ \mathit{alphabet}) \subseteq \mathit{knownActions} \\ \hline
\end{array}$$

2. The general state: this schema represents the general state information of part_activity1. The constraints in this schema relate the observations from the different sub-states: only known EventClasses can be allocated to an Alphabet Class; the known Resource Classes are a subset of the known ProcessClasses; the known Resources are a subset of the known Processes; only Events of known Event Classes can be created, and only Processes of known ProcessClasses can be created.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PartActivity1State}} \\ \hline
\mathit{EventClassState} \\
\mathit{ProcClassState} \\
\mathit{ResClassState} \\
\mathit{ProcState} \\ \hline
\bigcup(\mathrm{ran}\ \mathit{AlphabetClass}) \subseteq \mathit{knownEventClasses} \\
\\
\mathit{knownResClasses} \subseteq \mathit{knownProcClasses} \\
\\
\mathrm{ran}\ \mathit{classofEvent} \subseteq \mathit{knownEventClasses} \\
\mathrm{ran}\ \mathit{classofProcess} \subseteq \mathit{knownProcClasses} \\ \hline
\end{array}$$

3. The initial sub-states: there are four initial sub-states corresponding to each of the sub-states. The initial state of *EventClassState* is defined using a library schema *Init–Element–State[X]*:

$$\begin{array}{l}
\mathit{Init\text{–}EventClassState} \mathit{Init\text{–}Element\text{–}State}[\mathit{EVCLASS}] \\
\qquad\qquad [\ \mathit{knownEventClasses}/\mathit{knownElements}, \\
\qquad\qquad\qquad \mathit{EventClassDesc}/\mathit{ElementDesc}\ ]
\end{array}$$

which is equivalent to:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Init\text{–}EventClassState}} \\ \hline
\mathit{EventClassState} \\ \hline
\mathit{knownEventClasses} = \emptyset \\
\mathit{EventClassDesc} = \emptyset \\ \hline
\end{array}$$

The other initial sub-states are defined similarly.

4. The general initial state: part_activity1's initial state is comprised of the four initial sub-states.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Init\text{–}PartActivity1State}} \\ \hline
\mathit{PartActivity1State} \\
\\
\mathit{Init\text{–}EventClassState} \\
\mathit{Init\text{–}ProcessClassState} \\
\mathit{Init\text{–}ResourceClassState} \\
\mathit{Init\text{–}ProcessState} \\ \hline
\end{array}$$

5. Actions: we now give an example of an action belonging to this part_activity known as *CreateProcess*. This operation creates an instance of a ProcessClass. When a Process is created a new set of Events must be created by oDM. One new Event is created for each EventClass in the Alphabet Class of the ProcessClass of the Process being created. The pre-conditions for the operation are that the ProcessClass is known and the new Process is unknown. The first schema describes the role of the designer and the second the role of the CASE tool supporting oDM. In terms of the actions of the framework this is a $create$ action followed by a $derive$ action.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Create\text{–}Process}} \\ \hline
\oslash\ \mathit{ProcState} \\
\\
\mathit{pc}? : \mathit{PROCCLASS} \\
\mathit{p}? : \mathit{PROCESS} \\ \hline
\mathit{pc}? \in \mathit{knownProcClasses} \\
\mathit{p}? \notin \mathit{knownProcesses} \\
\\
\mathit{classofProcess}' = \mathit{classofProcess} \cup \{\ \mathit{p}? \mapsto \mathit{pc}?\ \} \\
\mathit{knownProcesses}' = \mathit{knownProcesses} \cup \{\ \mathit{p}?\ \} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Derive\text{–}Events}} \\ \hline
\oslash\ \mathit{ProcState} \\
\\
\mathit{pc}? : \mathit{PROCCLASS} \\
\mathit{p}? : \mathit{PROCESS} \\ \hline
\exists\, f : \mathit{EVENT} \pinj \mathit{EVCLASS} \\
\qquad \mid \#(\mathrm{dom}\ f) = \#(\mathit{AlphabetClass}(\mathit{pc}?)) \ \wedge \\
\qquad (\mathrm{dom}\ f \cap \mathit{knownActions}) = \emptyset \ \wedge \\
\qquad \mathrm{ran}\ f = \mathit{AlphabetClass}(\mathit{pc}?) \bullet \\
\qquad \mathit{alphabet}' = \mathit{alphabet} \cup \\
\qquad\qquad \{\ \mathit{p}? \mapsto (\mathrm{dom}\ f)\ \} \ \wedge \\
\qquad \mathit{classofEvent}' = \mathit{classofEvent} \cup f \ \wedge \\
\qquad \mathit{knownActions}' = \mathit{knownActions} \cup (\mathrm{dom}\ f) \\ \hline
\end{array}$$

In Z the successful operation is defined as follows:

$$\begin{array}{l}
\mathit{ReportSuccess\text{–}CreateProcess} \mathrel{\widehat{=}} [\, \mathit{report}! : \mathit{REP} \mid \\
\qquad \mathit{report}! = \mathsf{succ}(\mathtt{Create}, \mathtt{Process})\,] \\
\mathit{CreateProcess\text{–}OK} \mathrel{\widehat{=}} \mathit{Create\text{–}Process} \\
\quad \land\ \mathit{Derive\text{–}Events}\ \land\ \mathit{ReportSuccess\text{–}CreateProcess}
\end{array}$$

It can be seen that defining the *CreateProcess–OK* operation in terms of a create action followed by a derive action makes clear the distinction between information provided by the designer and that automatically derived during application of oDM. Another advantage of modularising our general state schemas into sub-states is that, since typically an operation will effect just one of the sub-state schemas, there will be no state change to the other components of the general part_activity state schema. In this operation, for example, the only sub-state which is affected is the *ProcState*; there is no state change to any of the other three modular components.

6. Terminal sub-states and general state: we first define the terminal sub-states for each of the sub-state schemas of part_activity1 by instantiation of the relevant generic schema

$$\begin{array}{l}
\mathit{Term\text{–}EventClassState} \mathrel{\widehat{=}} \mathit{Term\text{–}Element\text{–}State} \\
\quad [\mathit{EVCLASS}][\mathit{knownEventClasses}/\mathit{knownElements}, \\
\qquad\qquad \mathit{EventClassDesc}/\mathit{ElementDesc}\,]
\end{array}$$

which is equivalent to the following schema which states that all EventClasses have been described.

$$\begin{array}{|l}
\hline
\mathit{Term\text{–}EventClassState} \\
\mathit{EventClassState} \\
\hline
\mathsf{dom}\ \mathit{EventClassDesc} = \mathit{knownEventClasses} \\
\hline
\end{array}$$

The other terminal sub-states are defined similarly and the final terminal schema which specifies the completion of part_activity1 in oDM is given as follows. The schema constraints relate the different artifact_variables from the different terminal sub-states. In this case, all known EventClasses have been allocated to at least one Alphabet Class and there is at least one instance of each EventClass and ProcessClass.

$$\begin{array}{|l}
\hline
\mathit{Terminal\text{–}PartActivity1State} \\
\mathit{PartActivity1State} \\
\\
\mathit{Term\text{–}EventClassState} \\
\mathit{Term\text{–}ProcessClassState} \\
\mathit{Term\text{–}ResourceClassState} \\
\mathit{Term\text{–}ProcessState} \\
\hline
\bigcup(\mathsf{ran}\ \mathit{AlphabetClass}) = \mathit{knownEventClasses} \\
\\
\mathsf{ran}\ \mathit{classofEvent} = \mathit{knownEventClasses} \\
\mathsf{ran}\ \mathit{classofProcess} = \mathit{knownProcClasses} \\
\hline
\end{array}$$

Typically, on completion of a part_activity further 'derive' actions take place. For example, the known resources of the design are compiled automatically from part_activity1's terminal state as follows:

$$\begin{array}{|l}
\hline
\mathit{SystemDeriveFromPartActivity1State} \\
\mathit{Terminal\text{–}PartActivity1State} \\
\mathit{knownRes} : \mathbb{F}\ \mathit{PROCESS} \\
\hline
\mathit{knownRes} = \mathsf{dom}\ (\mathit{classofProcess} \rhd \mathit{knownResClasses}) \\
\hline
\end{array}$$

After this has been achieved, the designer moves on to the next part_activity and repeats the process outlined above. In general, whilst the designer has read access to previously defined information from preceding part_activities, they can not alter this information. Operation schemas on a general state thus take the following general form.

$$\begin{array}{|l}
\hline
\mathit{OperationOnCurrentPartActivityState} \\
\Delta\, \mathit{CurrentPartActivityState} \\
\Xi\ \mathit{Inference\text{–}From\text{–}PreviousPartActivityState} \\
\hline
\end{array}$$

Acknowledgements

Thanks to Colin Myers for comments on earlier drafts of this paper.

6 Conclusions

Although HPS are now widely available, little work has been done in trying to support SDM developers in producing methods and tools for the development of these systems. More importantly, we have not seen any attempt in systematically classifying and evaluating the SDMs currently available. This paper puts forward the first solution in this direction – a formal framework for describing SDMs for HPS. The framework can be used informally to classify SDM and tools but one of its important properties is to allow the derivation of a formal specification of a given methodology. By applying this approach to an existing methodology known as oDM we have encountered a number of benefits to the method. First, as direct result of the Z specification, the method has been made more precise, rigorous and complete. Informal aspects of oDM have been isolated and refined, inconsistencies have been removed and new notions introduced. In addition, there is now an increased understanding and representation of the relationships between components of the SDM. Further, the role of the designer, the role of a supporting CASE tool and the interaction between the designer and the tool can be fully defined within the framework.

We have found Z an ideal means of formally presenting a design methodology. Through the use of an abstract specification we do not restrict a specifier to any particular mathematical model; rather it provides a general mathematical framework within which particular systems can be defined and contrasted.

The next step in the evaluation of the framework will be to apply it to the SEPP project (Software Engineering for Parallel Processing) funded by the EEC and partially developed at the University of Westminster. The methodology behind the SEPP project is very informal but the software development environment it proposes covers many of the activities included in our software process model. We believe that the framework will help in formalising the relationships and roles of the tools within the environment, thereby formally structuring the methodology.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[2] B. I. Blum. A taxonomy of software development methods. *Communications of ACM*, 37(11):82–94, November 1994.

[3] R. Conradi, C. Fernstrom, and A. Fuggetta. A conceptual framework for evolving software processes. *ACM SIGSOFT: Software Engineering Notes*, 18(4):26–35, October 1993.

[4] P. Dauphin. Combining functional and performance debugging of parallel and distributed systems based on model-driven monitoring. In *2nd Euromicro Workshop on Parallel and Distributed Processing*, pages 463–470. IEEE Computer Society Press, 1994.

[5] A. M. Davis, E. H. Bersoff, and E. R. Comer. A strategy for comparing alternative software life cycle models. *IEEE Software*, 14(10):1453–1461, October 1988.

[6] M d'Inverno and J. Crowcroft. Design, specification and implementation of an interactive conferencing system. In *Proceedings of IEEE Infocom, Miami, USA. Published IEEE*, 1991.

[7] M. d'Inverno and M. Priestley. Structuring a Z specification to provide a unifying framework for hypertext systems. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: 9th International Conference of Z Users, Lecture Notes in Computer Science*, pages 81–102, Heidelberg, Springer-Verlag, 1995.

[8] A. Ferscha and J. Johnson. N-MAP: A virtual processor discrete event simulation tool for performance prediction in the CAPSE environment. In *28th Annual Hawaii International Conference on Systems Sciences*, pages 276–285. IEEE Computer Society Press, 1995.

[9] D. Harel. On visual formalisms. *Communications of ACM*, 31(5):514–530, May 1988.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] P. Howells, S. Goldsack, and B. Quirk. Designing from Modal Action Logic (MAL). Technical report, FOREST Project, Imperial College of Science & Technology, 1988.

[12] Inmos. *Occam2: Toolset User Manual – Part 1 (User Guide and Tools)*, 1991.

[13] L. H. Jamieson. Characterizing parallel algorithms. In L.H. Jamieson, D. Gannon, and R. J. Douglas, editors, *The characteristics of Parallel Algorithms*, pages 65–100. The MIT Press, 1987.

[14] I. Jelly and I. Gorton. Software engineering for parallel systems. *Information and Software Technology*, 36(7):379–380, 1994.

[15] G. R. R. Justo. A graphical approach to performance-oriented development of parallel programs. In *Second International Conference on High Performance Computing*. Tata McGraw Hill, 1995.

[16] G. R. R. Justo. A rigorous method for the constructive design of parallel and distributed programs. In *28th Annual Hawaii International Conference on Systems Sciences*, pages 319–328. IEEE Computer Society Press, January 1995.

[17] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254-260, AAAI Press / MIT Press, 1995.

[18] M. Luck and M. d'Inverno. Structuring a Z specification to provide a formal framework for autonomous agent systems. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: 9th International Conference of Z Users, Lecture Notes in Computer Science*, pages 47–62, Heidelberg, Springer-Verlag, 1995.

[19] Luqui. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, May 1989.

[20] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *2nd Int. Workshop on Configurable Distributed Systems*. IEEE Computer Society Press, March 1994.

[21] D. McCraken and M. Jackson. Life cycle concept considered harmful. *ACM SIGSOFT: Software Engineering Notes*, 7(2):29–32, April 1982.

[22] D. A. Padua and M. J. Wolfe. Advanced compiler optimisations for supercomputers. *Communications of ACM*, 29(12):1184–1201, December 1986.

[23] X. Song. A framework for understanding the integration of design methodologies. *ACM SIGSOFT: Software Engineering Notes*, 20(1):46–54, January 1995.

[24] X. Song and L. J. Osterweil. Towards objective, systematic design-methods comparisons. *IEEE Software*, 9:43–53, May 1992.

[25] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[26] R. van Rein. An object oriented framework for mapping concurrent applications to parallel and distributed architectures, 1994.

[27] G. Wirtz. Modularization, re-use and testing for parallel message-passing programs. In *28th Annual Hawaii International Conference on Systems Sciences*, pages 299–308. IEEE Computer Society Press, January 1995.