Formal Aspects of Computing (1993) 3: 1–000

A Formal Semantics of Data Flow Diagrams

Peter Gorm Larsen,[1] Nico Plat[2] and Hans Toetenel[2]

[1] IFAD, Odense, Denmark.
[2] Delft University of Technology, Faculty of Technical Mathematics and Informatics, Delft, The Netherlands. Nico Plat is currently with CAP Volmac, Utrecht, The Netherlands.

Keywords: Data flow diagrams; VDM; Formal semantics

Abstract. This paper presents a formal semantics of data flow diagrams as used in Structured Analysis, based on an abstract model for data flow transformations. The semantics consists of a collection of VDM functions, transforming an abstract syntax representation of a data flow diagram into an abstract syntax representation of a VDM specification. Since this transformation is executable, it becomes possible to provide a software analyst/designer with two 'views' of the system being modeled: a graphical view in terms of a data flow diagram, and a textual view in terms of a VDM specification. In this paper emphasis is on the motivation for the choices made in the transformation. The main aspects of the transformation itself are described using annotated VDM functions with some examples.

1. Introduction

The introduction of formal methods in industrial organizations may become easier if these methods can be used alongside the more widely used conventional techniques for software development, such as 'structured methods'. Structured methods are methods for software analysis and design, based on the use of heuristics for making analysis and design decisions. They provide a relatively well-defined path, often in a cookbook-like fashion (hence the term 'structured' methods), starting from the analysis of software requirements and ending at system coding. The design notations used are usually graphical and have no formal

Correspondence and offprint requests to: Peter Gorm Larsen, IFAD, Forskerparken 10, DK-5230 Odense M, Denmark. E-mail: peter@ifad.dk

basis. In that sense structured and formal methods can be regarded as complementary. It is often suggested that the informal graphical notations as provided by structured methods are intuitively appealing to software analysts/designers. Therefore, a combined structured/formal method may not only increase the understanding of the use of formal methods in the software process, but also may increase the acceptability of formal methods to these people.

Our work in this area so far has concentrated on combining *Structured Analysis (SA)* [You75, DeM79, GS79] with the *Vienna Development Method (VDM)* [BJ82, Jon90]; we provide a brief introduction to SA, but we refer to text books such as [Jon90] and [AI91] for an introduction to VDM. We think that a well-integrated combination of notations can be achieved by using *data flow diagrams (DFDs)* – which we consider to be the main design notation of SA – as a graphical view of the system and VDM as a textual view. These different views emphasize different aspects of the specified system: the DFD graphical view focuses on an overview of the *structure* of the system, whereas the VDM textual view focuses on the detailed *functionality* of the system. The base of a combined structured/formal method consists of a formally defined relation between the structured method and the formal method. In [PvKP91] we describe several approaches to modeling DFDs using the VDM-SL specification language [—92, Daw91]. In this paper we discuss one such particular model in more detail, thus essentially providing a 'formal semantics' of DFDs. A discussion on the methodological aspects of the approach can be found in [LvKP+93].

The remainder of this paper is organized as follows. In the following section a brief introduction to SA is given, focusing on the use of DFDs. In Section 3 we describe our strategy for transforming DFDs into VDM specifications, paying attention to the limitations of our approach. The main part of this paper is Section 4, in which the formal transformation from DFDs to VDM is presented. First, we describe the main aspects of an abstract syntax representation of DFDs (the abstract syntax representation we use for VDM specifications is the same as the one used in [—92][1]), and then we describe the formal transformation itself. Emphasis is put on the motivation for the choices made in the transformation. The main aspects of the transformation itself are described using VDM functions together with a number of examples. Given the limited size of a research paper like this, we have chosen to limit the description of the formal aspects of the transformation as much as possible. Therefore, in some situations it is necessary to rely on an intuitive understanding of what a function does. Furthermore, some functions have been somewhat simplified so that attention can be focused on the relevant aspects only. The complete transformation has been syntax-checked, type-checked and tested using the IFAD VDM-SL Toolbox [Las93]; this has given us confidence that the transformation we have defined is a reasonable one. The complete specification can be freely obtained by contacting any of the authors. Finally, in Section 5 we give an overview of related work on formal semantics for DFDs, and present some conclusions and ideas for future work in this area.

[1] To be precise, the abstract syntax used for VDM specifications is the one called 'Outer Abstract Syntax' in [—92]; a lack of knowledge about this Outer Abstract Syntax does not affect the understanding of this paper, however.

2. Overview of Structured Analysis

Structured Analysis (SA) [You75, DeM79, GS79] is one of the most widely used methods for software analysis. Often it is used in combination with *Structured Design (SD)* [CY79]; the resulting combination is called SA/SD. The approach to analysis taken in SA is to concentrate on the functions to be carried out by the system, using data flow abstraction to describe the flow of data through a network of transforming processes, called data transformers, together with access to data stores. Such a network, which is the most important design product of SA, is called a *data flow diagram (DFD)*. The original version of SA was meant to be used to model sequential systems. A DFD is a *directed graph* consisting of elementary building blocks. Each building block has a graphical notation (figure 1).

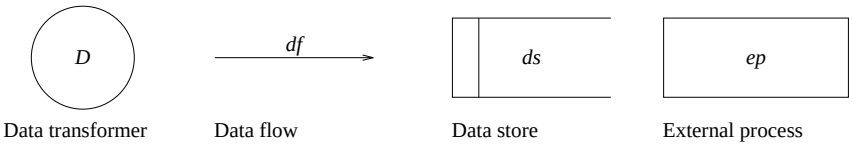

Fig. 1. Elementary building blocks of a DFD

Through the years several dialects have evolved and extensions have been defined (e.g. SSADM [LN86] and SA/RT [WM85]), but we limit ourselves to DFDs with a sequential model and a small number of building blocks:

- *Data transformers.* Data transformers denote a transformation from (an arbitrary number of) input values to (an arbitrary number of) output values, possibly with side effects.
- *Data flows.* Data flows are represented as arrows, connecting one data transformer to another. They represent a flow of data between the data transformers they connect. The flow of data is unidirectional in the direction of the arrow.
- *Data stores.* Data stores provide for (temporary) storage of data.
- *External processes.* External processes are processes that are not part of the system but belong to the outside world. They are used to show where the input to the system is coming from and where the output of the system is going to.

DFDs are used to model the *information flow* through a system. As such they provide a limited view of the system: in their most rudimentary form they neither show the control flow of the system nor any timing aspects. Therefore, DFDs are often combined with data dictionaries, control flow diagrams, state transition diagrams, decision tables and mini-specifications to provide a comprehensive view of all the aspects of the system.

The process of constructing a DFD is an iterative process. Initially, the system to be designed is envisaged as one large data transformer, getting input from and providing output to external processes. This initial, high-level DFD is called a *context diagram*. The next step is the *decomposition* of the context diagram into a network of data transformers, the total network providing the same functionality as the original context diagram. This process is repeated for each data transformer until the analyst/designer considers all the data transformers in the

DFD to be primitive, i.e. each data transformer performs a simple operation that does not need to be further decomposed. We call such a collection of DFDs, describing the same system but at different levels of abstraction, a *hierarchy of DFDs*.

3. Approach to the Transformation

Before presenting the formal transformation from DFDs to VDM we first explain the underlying strategy for the transformation and the limitations imposed upon the DFDs to make our transformation valid.

3.1. Underlying strategy

The starting point for our transformation is the work presented in [PvKP91], in which the general properties of two transformations from DFDs to VDM constructs are discussed. The main difference between these two transformations is the way data flows are modeled: in the first transformation they are modeled as (infinitely large) queues, in the second transformation they are modeled as operations combining the two data transformers connected by the data flow. The advantage of the latter transformation is that a more abstract interpretation of DFDs can be achieved, because the transformation solely focuses on modeling the information flow through a DFD. This is also the reason for choosing this transformation as the basis for the transformation described in this paper. One simplification with respect to the transformation described in [PvKP91] is that the latter is more general because the order in which the 'underlying' operations are called is left unspecified (i.e. it is loosely specified), which makes the operation modeling the data flow rather complicated. In this paper, however, we are dealing with purely sequential systems, and therefore we can assume that data flows between two data transformers are 'direct' in the sense that the data transformer that uses the data flow as input cannot be called before the data transformer that uses the data flow as output.

3.2. Transformation of DFD building blocks

When providing a formal semantics for DFDs it is important to decide whether the DFD is intended to model a concurrent system or a sequential system. More recent versions of SA (like *SA/RT* [WM85]) include concurrency and can be used to develop real-time systems. However, originally SA was intended for the development of information systems implemented in traditional imperative programming languages. In that situation it seems natural to interpret the data transformers as *functions* or *operations* which, given input data, sequentially perform computations and produce output data. If the data flow diagrams are used to model a concurrent system it is more natural to interpret data transformers as *processes*, possibly executing in parallel. Since we restrict ourselves to sequential systems we model data transformers as *VDM operations*[2].

[2] Data transformers neither having access to data stores nor being connected to external processes can also be modeled as *VDM functions*. In our approach only VDM operations are used

To ensure that the structure of the VDM specification resembles the structure of the DFD, we group the operations modeling data transformers at the same level in a hierarchy of DFDs together in 'modules'[3] importing the necessary types and operations needed for the data transformers (figure 2).

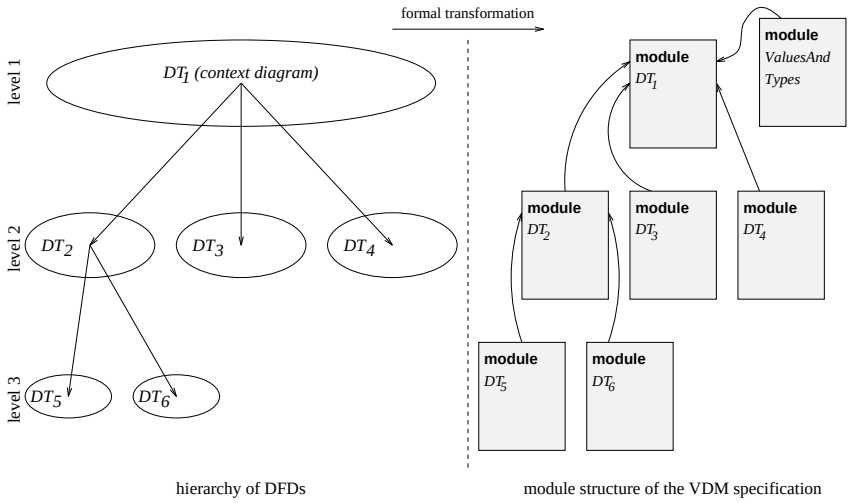

Fig. 2. Transformation of a hierarchy of DFDs into a VDM module structure (example)

External processes can be considered as processes 'executing' in parallel with the specified system. In our approach we model the data flows from and to external processes as *state components* in the VDM specification. This is a minor difference with the transformation presented in [PvKP91], in which external processes are regarded as part of the system and are therefore modeled as VDM operations in the same way as data transformers.

Data stores are modeled as VDM *state components*. This corresponds to the fact that data transformers (which can be used to access and change data stores) are modeled as VDM operations, the constructs in VDM-SL that can access and change state components.

We envisage *data flows* as constructs which can combine two data transformers by providing communication facilities between these two data transformers. A data flow is, therefore, modeled as an operation calling the operations that model the two data transformers connected by the data flow. In this way a process of combining data transformers can be started during which in each step two data transformers (connected by a data flow) are integrated into a higher level data transformer, finally resulting in the context diagram.

Generalizing this approach, we have chosen to combine all the data transformers in a DFD into a higher level data transformer in *one* step. The data transformer constructed in this way is modeled as a VDM operation.

---

because we want each different type of construct in a DFD to be mapped to (semantically) the same construct in VDM. VDM functions and VDM operations (without side-effects) semantically differ in the way *looseness* is interpreted (see [LAMB89]).

[3] VDM-SL as described in [—92] has no structuring mechanism. The structuring mechanism we used is based on a proposal by Bear [Bea88]. The constructs we use are simple so that an intuitive interpretation suffices.

3.3. Limitations imposed upon the DFDs

Besides restricting the expressibility of the kind of DFDs for which we are able to provide semantics to sequential systems, we assume that:

- Data flows not connected to an external process must form an acyclic graph at each level in the hierarchy of DFDs. This is necessary because in our transformation we provide both explicit VDM specifications as well as implicit VDM specifications as models for DFDs. Allowing general cyclic DFDs would make the transformation into an explicit VDM specification impossible. The restriction furthermore simplifies the transformation of DFDs into implicit VDM specifications. In Section 4.2.2 we come back to this restriction in more detail.
- There is a one-to-one mapping between the input to the system and the output from the system. One-to-many mappings and many-to-one mappings are a common problem when interpreting DFDs, described in more detail in [Ala88][4]. However, we are not entirely satisfied with the solution proposed by Alabiso, and since in our experience most of the DFDs with one-to-many or many-to-one mappings should be regarded as design products and not as specification products, we feel that a restriction to one-to-one mappings is not a serious one for our purpose. Alternatively, the analyst may supply a mini-specification for each non-primitive data transformer not obeying the restriction of a one-to-one mapping between input and output.
- To simplify the formal description the data flows must have unique names at each level in the hierarchy of DFDs.

4. Transformation from DFDs to VDM

This section provides a presentation of the transformation from DFDs to VDM. It takes an abstract syntax representation of the DFDs which is presented in the first subsection and yields the abstract syntax of a modular VDM specification. The constructs from the abstract syntax for VDM-SL which are used below should be directly understandable without examining the VDM-SL standard [—92]. The last subsection presents an overview of the actual transformation functions.

4.1. The Abstract Syntax

The SA concepts used in the transformation process are a hierarchy of data flow diagrams ($HDFD$), a data dictionary (DD), and a collection of uniquely identified mini-specifications (MSs). The types of all data flows in the data flow diagrams must be defined in the data dictionary. Besides this, the signature of the top-level DFD must conform to its topology.

$$SA = HDFD \times DD \times MSs$$

[4] In [Ala88] this problem is called *I/O uncohesiveness*. I/O uncohesiveness occurs if either a data transformer must consume several pieces of input data before generating output data, or if a data transformer generates pieces of output independently of all other inputs and outputs. Alabiso describes a solution called 'the burial method', centered around the generation of terminator symbols which indicate that 'something is missing'.

$$\textsf{inv}\ mk\text{-}(hdfd, dd, \text{-}) \triangleq FlowTypeDefined(hdfd, dd) \wedge TopLevelSigOK(hdfd)$$

The hierarchy of data flow diagrams is recursively defined. Each *HDFD* has a name, an unordered collection of data stores used in the DFD, a description of its topology, a collection of uniquely identified data transformers that are further decomposed as *HDFD*s, and a description of the signatures of all the data transformers.

The invariant for *HDFD* ensures that the signatures of the data transformers (and the DFD as a whole) are consistent with the topology and the data stores, and that all the DFDs – which are further decomposed – are defined.

$$HDFD = DFDId \times DSs \times DFDTopo \times DFDMap \times DFDSig$$

$$\textsf{inv}\ mk\text{-}(id, dss, dfdtop, dfdmap, dfdsig) \triangleq DFDSigConsistent(id, dfdtop, dss, dfdmap, dfdsig) \wedge LowerLevelUsed(dfdtop, dfdmap)$$

The topology of a DFD is a collection of uniquely identified data flows. Each data flow is directed from one data transformer to another. The data transformers can either be further decomposed or they can be primitive. The invariant requires that the data flow connects two data transformers and that the topology of the internal connections is acyclic.

$$DFDTopo = FlowId \xrightarrow{m} Flow$$

$$\textsf{inv}\ dfdtopo \triangleq \textsf{let}\ top = \{flow \mid flow \in \textsf{rng}\ dfdtopo \cdot InternalFlow(flow)\}\ \textsf{in}\ NotRecursive(top) \wedge \forall flow \in \textsf{rng}\ dfdtopo \cdot FlowConnectOK(flow)$$

4.2. The Transformation Functions

In this section we first present the top-level function and the main function for creating a collection of definitions. The transformation functions are able to compose the data transformers using either an implicit style or an explicit style. These two styles are dealt with in more detail in the last two subsections. This is done by first looking at a few simple examples and then presenting the actual definition of this transformation.

4.2.1. The Top-level Functions

The top-level function, which transforms a hierarchy of data flow diagrams (an *HDFD*), also takes as arguments the mini-specifications supplied by the user and the specification style in which the operations are to be generated.

$$
\begin{array}{l}
TransHDFD : HDFD \times MSs \times (\text{EXPL} \mid \text{IMPL}) \to Module\text{-set} \\
\\
TransHDFD\,(hdfd, mss, style) \triangleq \\
\quad \mathsf{let}\ mainmod \quad\quad\quad\quad = MakeDFDModule\,(hdfd, mss, style), \\
\quad\quad mk\text{-}(\text{-}, \text{-}, \text{-}, dfdmap, \text{-}) = hdfd, \\
\quad\quad mods \quad\quad\quad\quad = \bigcup \{ TransHDFD\,(dfd, mss, style) \mid \\
\quad\quad\quad\quad\quad\quad\quad\quad dfd \in \mathsf{rng}\ dfdmap \}\ \mathsf{in} \\
\quad \{mainmod\} \cup mods
\end{array}
$$

For each module the interface and the definitions must be created by means of *MakeDFDModule*; we limit ourselves here to the definitions-part of a module, however. If the DFD contains data stores, the body will contain a corresponding state definition. If the DFD contains data transformers that are not further decomposed, the body also contains definitions for these. Finally the module will always contain a definition of the operation that describes the functionality of that DFD.

$$
\begin{array}{l}
MakeDefinitions : \\
\quad DFDId \times DSs \times DFDTopo \times DFDSig \times MSs \times \\
\quad (\text{EXPL} \mid \text{IMPL}) \to Definitions \\
\\
MakeDefinitions\,(dfdid, dss, dfdtopo, dfdsig, mss, style) \triangleq \\
\quad \mathsf{let}\ st' \quad\quad = MakeState\,(dfdid, dss, CollectExtDFs\,(dfdtopo)), \\
\quad\quad msdescs = MakeMSDescs\,(dfdsig, mss), \\
\quad\quad dfdop \quad\quad = MakeDFDOp\,(dfdid, dfdtopo, dfdsig, style)\ \mathsf{in} \\
\quad \mathsf{if}\ st' = \mathsf{nil} \\
\quad \mathsf{then}\ \{dfdop\} \cup msdescs \\
\quad \mathsf{else}\ \{st', dfdop\} \cup msdescs
\end{array}
$$

4.2.2. Generation of Implicit Operations

An operation describing the functionality of a DFD uses the operations for the lower-level DFDs. The combination that must be constructed depends upon the topology of the DFD. Whenever a data transformer receives data from another data transformer through a data flow (in the same DFD) this dependency must be incorporated in the combination, by using the output value from the first data transformer (and possibly changed state component(s)) as input for the second data transformer. However, since a data transformer in principle is a loose construct it is necessary when generating pre- and post-conditions to take this possible looseness into account. This is done by specifying that there must exist an output value (and possibly one or more changed state values) such that the post-condition of the first data transformer is fulfilled and then use this value (or values) for the data transformer which depends upon the first one (see e.g. [PvKP91]).

By means of three small examples we illustrate the issues to be considered when describing the functionality of a DFD as a whole.

Example 1

Consider the DFD in figure 3. It is a simple DFD consisting of two data transformers P and Q, each having one input data flow (a and b respectively) and one output data flow (b and c respectively). Q receives data from P and thus Q depends on P. When this DFD is intended to model a sequential system it is

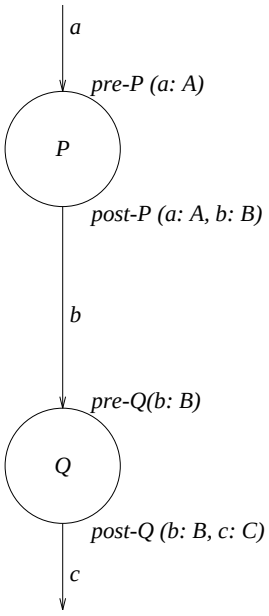

Fig. 3. DFD for example 1

obvious that P must be executed before Q can be executed. This dependency between P and Q also can be found in the pre- and post-condition of the composite DFD:

$$
\begin{array}{l}
PQ\,(a:A)\;\; c:C \\
\mathsf{pre}\;\; \exists\, b:B \cdot \\
\qquad \textit{pre-}P(a) \wedge \\
\qquad \textit{post-}P(a,b) \wedge \\
\qquad \textit{pre-}Q(b) \\
\mathsf{post}\; \exists\, b:B \cdot \\
\qquad \textit{pre-}P(a) \wedge \textit{post-}P(a,b) \wedge \\
\qquad \textit{pre-}Q(b) \wedge \textit{post-}Q(b,c)
\end{array}
$$

It is necessary to quote the post-condition[5] of P to produce a value that must satisfy the pre-condition of Q. Since P may be loosely specified there may be several values satisfying the post-condition of P given some argument a. However, since only some of these values might satisfy the pre-condition of Q an

[5] 'Quoting' pre- and post-conditions of (implicitly defined) functions and operations is a VDM technique to 'invoke' other functions or operations from within a pre- or post-condition (i.e. a predicate): each implicitly defined function or operation f has associated boolean *functions* *pre-f* and *post-f* which, given the appropriate arguments, yield *true* if the pre- or post-condition respectively of f holds for those arguments, and *false* otherwise. A quoted pre-condition of an operation takes the input arguments of the operation and the state components used by the operation as its arguments. A quotation of a post-condition of an operation first takes the input arguments of the operation, then some arguments representing the values of the state components before the operation is executed, the output result of the operation, and finally the new state components (only those to which the operation has write access).

existential quantification over this 'internal data flow', b, is necessary. Alternative solutions can be envisaged, differing in the strength of the constraints put upon the combination.
□

Example 2

Example 1 is now expanded by introducing a data store that both data transformer P and data transformer Q have write access to. This DFD is given in figure 4. The data store ds is – as has been mentioned – interpreted as a state component.

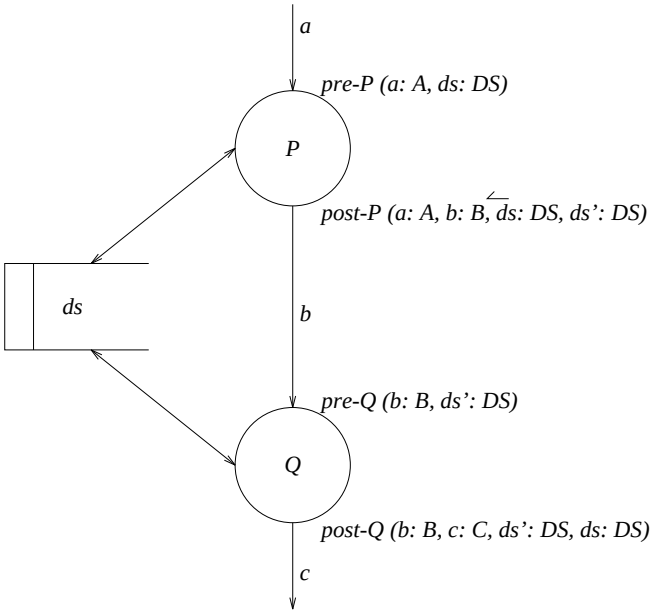

Fig. 4. DFD for example 2

This composite DFD can be specified by the following implicit definition:

$$
\begin{array}{l}
PQ_{DS}\,(a:A)\;\; c:C \\
\textsf{ext wr}\;\; ds:DS \\
\textsf{pre}\;\; \exists\, b:B, ds':DS \cdot \\
\qquad pre\text{-}P(a, ds) \wedge \\
\qquad post\text{-}P(a, ds, b, ds') \wedge \\
\qquad pre\text{-}Q(b, ds') \\
\textsf{post}\; \exists\, b:B, ds':DS \cdot \\
\qquad pre\text{-}P(a, \overleftarrow{ds}) \wedge post\text{-}P(a, \overleftarrow{ds}, b, ds') \wedge \\
\qquad pre\text{-}Q(b, ds') \wedge post\text{-}Q(b, ds', c, ds)
\end{array}
$$

It is necessary to introduce an intermediate state component, ds', which holds the value of ds in between execution of the different data transformers, P and

Q. This situation occurs when several data transformers are allowed to modify the same data store.

In addition, this example illustrates another technicality that must be taken into account in the transformation from DFDs to VDM. The value of the state component, *ds*, before activation of the operation is referred to differently inside the pre-condition (as *ds*) and the post-condition (as $\overleftarrow{ds}$). When a pre- or post-condition (using an old state value) is quoted it is necessary to supply information about whether it was quoted inside a pre-condition or inside a post-condition. □

Example 3

The DFD from example 2 is now expanded by adding an extra data transformer, R, which also modifies data store *ds*, but otherwise is not connected to the two other data transformers (P and Q). The DFD is given in figure 5.

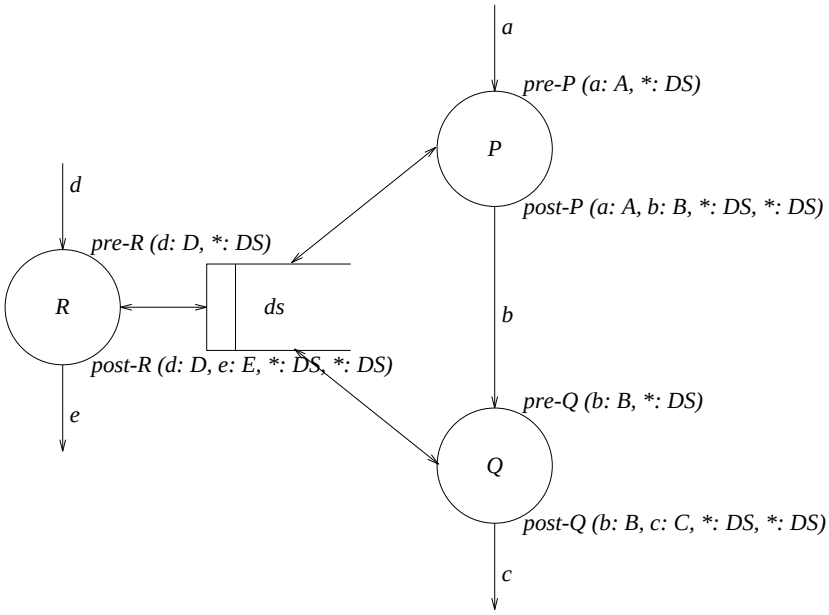

Fig. 5. DFD for example 3

Although the DFD at first sight still looks rather simple, it turns out that the VDM specification for the DFD is quite complicated. The DFD is illustrative for the situation in which the *writer* of the DFD may understand it differently than the *reader* of the DFD. The ambiguity comes from the fact that nothing is said about in which order the three data transformers should modify the data store. Maybe it is not important, but maybe it is essential that one specific execution order is chosen in the implementation. The notation '$*:DS$' (in the figure) means that a value of type DS will be used at this point, but we don't know exactly *which* value that will be. Consider P and R. One of them uses the old value of *ds* in the quotation of its post-condition, but we don't know which one because

that depends on the execution order. The possible execution orders are visible in the generated VDM specification.

The following implicit definition of the composite DFD can be generated:

$$
\begin{array}{l}
PQR_{DS}\,(a:A, d:D)\ r:C\times E \\
\textsf{ext wr}\ ds:DS \\
\textsf{pre}\ \exists\, b:B, c:C, e:E, ds', ds'':DS\cdot \\
\qquad (pre\text{-}R(d, ds)\wedge post\text{-}R(d, ds, e, ds')\wedge \\
\qquad\ pre\text{-}P(a, ds')\wedge post\text{-}P(a, ds', b, ds'')\wedge pre\text{-}Q(b, ds''))\vee \\
\qquad (pre\text{-}P(a, ds)\wedge post\text{-}P(a, ds, b, ds')\wedge \\
\qquad\ pre\text{-}R(d, ds')\wedge post\text{-}R(d, ds', e, ds'')\wedge pre\text{-}Q(b, ds''))\vee \\
\qquad (pre\text{-}P(a, ds)\wedge post\text{-}P(a, ds, b, ds')\wedge \\
\qquad\ pre\text{-}Q(b, ds')\wedge post\text{-}Q(b, ds', c, ds'')\wedge pre\text{-}R(d, ds'')) \\
\textsf{post let}\ (c, e) = r\ \textsf{in} \\
\quad \exists\, b:B, ds', ds'':DS\cdot \\
\qquad (pre\text{-}R(d, \overleftarrow{ds})\wedge post\text{-}R(d, \overleftarrow{ds}, e, ds')\wedge pre\text{-}P(a, ds')\wedge \\
\qquad\ post\text{-}P(a, ds', b, ds'')\wedge pre\text{-}Q(b, ds'')\wedge post\text{-}Q(b, ds'', c, ds))\vee \\
\qquad (pre\text{-}P(a, \overleftarrow{ds})\wedge post\text{-}P(a, \overleftarrow{ds}, b, ds')\wedge pre\text{-}R(d, ds')\wedge \\
\qquad\ post\text{-}R(d, ds', e, ds'')\wedge pre\text{-}Q(b, ds'')\wedge post\text{-}Q(b, ds'', c, ds))\vee \\
\qquad (pre\text{-}P(a, \overleftarrow{ds})\wedge post\text{-}P(a, \overleftarrow{ds}, b, ds')\wedge pre\text{-}Q(b, ds')\wedge \\
\qquad\ post\text{-}Q(b, ds', c, ds'')\wedge pre\text{-}R(d, ds'')\wedge post\text{-}R(d, ds'', e, ds))
\end{array}
$$

The post-condition shows that there are three possible execution orders: $[P, Q, R]$, $[P, R, Q]$ and $[R, P, Q]$. The pre- and post-conditions defined above ensure that at least one possible execution order can be used. r is a new name, introduced to denote the output as a whole.
□

Below, we present the functions that are used to compose data transformers into implicit specifications. These functions illustrate how the problematic issues from the three examples above are dealt with.

First, consider the function *MakeDFDImplOp* that is used to create operations for DFDs in the implicit style. The signature of the given DFD is used to create the input parameter list, the possible output pair and the externals for the operation (by means of a few auxiliary functions) whereas the body of the operation is much more complicated to create, so this is further explained below.

$$
\begin{array}{l}
MakeDFDImplOp : DFDId\times DFDTopo\times DFDSig\to ImplOp \\
MakeDFDImplOp\,(dfdid, dfdtopo, dfdsig)\ \triangleq \\
\quad \textsf{let}\ mk\text{-}(in', out, st') = dfdsig\,(dfdid), \\
\qquad partpl = MakeInpPar\,(in'), \\
\qquad residtp = MakeOutPair\,(out), \\
\qquad ext' = MakeExt\,(st'), \\
\qquad body = MakeImplOpBody\,(dfdid, dfdtopo, dfdsig)\ \textsf{in} \\
\quad mk\text{-}ImplOp\,(OpIdConf\,(dfdid), partpl, residtp, ext', body)
\end{array}
$$

The function *MakeImplOpBody* is used to generate both the pre-condition and the post-condition of an implicit operation definition. To take intermediate data store values into account, *MakeImplOpBody* and its auxiliary functions use a map from state components to the current number of intermediate values ($intm$).

The map is initialized by mapping all state components to zero (indicating that no intermediate state values have yet been introduced)[6]. In addition, a map *maxm* with the same domain of state components is used to ensure that a post-condition uses the state after an operation as the last of a series of intermediate state components. Each state component in *maxm* is mapped to the number of data transformers in a partition having write access (and thus potentially introduce an intermediate state value) to that state component.

$$
\begin{array}{l}
MakeImplOpBody : DFDId \times DFDTopo \times DFDSig \to ImplOpBody \\
MakeImplOpBody\,(dfdid, dfdtopo, dfdsig) \triangleq \\
\quad \mathsf{let}\ intm = \{stid \mapsto 0 \mid \\
\qquad\qquad mk\text{-}(stid, \text{-}) \in CollectStIds\,(\mathsf{rng}\ dfdsig)\}, \\
\qquad maxm = \{stid \mapsto NoOfWr\,(\mathsf{rng}\ dfdsig, stid) \mid \\
\qquad\qquad mk\text{-}(stid, \text{-}) \in CollectStIds\,(\mathsf{rng}\ dfdsig)\}, \\
\qquad pre' = MakePreExpr\,(dfdid, dfdtopo, dfdsig, intm, maxm), \\
\qquad post' = MakePostExpr\,(dfdid, dfdtopo, dfdsig, intm, maxm)\ \mathsf{in} \\
\quad mk\text{-}ImplOpBody\,(pre', post')
\end{array}
$$

MakePostExpr is used to generate the post-condition of an implicit operation[7]. The body of the post-condition is created by means of the function *MakePostBody*. When a DFD has more than one output, these outputs are combined into a tuple expression. A new name for this tuple expression is created by generating a let-expression. *MakePostExpr* decides whether such a let-expression should be generated and if so a let-expression is put around the body.

$$
\begin{array}{l}
MakePostExpr : DFDId \times DFDTopo \times DFDSig \times IntM \times IntM \to Expr \\
MakePostExpr\,(dfdid, dfdtopo, dfdsig, intm, maxm) \triangleq \\
\quad \mathsf{let}\ mk\text{-}(\text{-}, out, st') = dfdsig\,(dfdid), \\
\qquad body = MakePostBody\,(out, st', dfdtopo, dfdsig, \\
\qquad\qquad\qquad\qquad intm, maxm)\ \mathsf{in} \\
\quad \mathsf{if}\ \mathsf{len}\ out \leq 1 \\
\quad \mathsf{then}\ body \\
\quad \mathsf{else}\ mk\text{-}LetExpr\,(MakePattern\,(out), ResultIdConf\,(), body)
\end{array}
$$

The function *MakePostBody* examines whether an existential quantification is needed[8], and if this is the case such a quantified expression is generated. The remaining part of the post-condition is generated by calling *MakePostPred*.

[6] The configuration function *StateVarIntConf* inserts a number of quotes corresponding to the number of the intermediate value, as it was done in the examples.

[7] The function *MakePreExpr* is essentially doing the same thing for pre-conditions, but it is slightly simpler.

[8] As can be seen in the examples above, quantification is necessary for data flows that are not direct input (or output) to a given DFD or if intermediate state values are needed.

$$
\begin{array}{l}
MakePostBody : \\
\quad FlowId^* \times State \times DFDTopo \times DFDSig \times IntM \times IntM \to Expr \\
MakePostBody\,(out, st', dfdtopo, dfdsig, intm, maxm) \triangleq \\
\quad \textsf{let } pred = MakePostPred\,(dfdtopo, dfdsig, intm, maxm), \\
\qquad fids = NeedsQuant\,(dfdtopo, dfdsig, \textsf{elems } out, \{\}) \textsf{ in} \\
\quad \textsf{if } QuantNecessary\,(out, st', fids, intm, maxm) \\
\quad \textsf{then } mk\text{-}ExistsExpr\,(MakeExistsBind\,(fids, st', intm, maxm, \textsc{post}), \\
\qquad\qquad pred) \\
\quad \textsf{else } pred
\end{array}
$$

The function *MakePostPred* is used to create the core part of the 'body' of the post-condition of an implicit operation. First, all possible orders of execution are determined, and for each execution order[9] a conjunction of quoted function applications are generated using the intermediate state values (this is done in *MakePostForEO*). The separate conjunctions are then combined in one large disjunction, in this way specifying that the implementor can choose either one of the execution orders to implement the DFD.

$$
\begin{array}{l}
MakePostPred : DFDTopo \times DFDSig \times IntM \times IntM \to Expr \\
MakePostPred\,(dfdtopo, dfdsig, intm, maxm) \triangleq \\
\quad \textsf{let } eos = ExecutionOrders\,(dfdtopo) \textsf{ in} \\
\quad DBinOp\,(\textsc{or}, \{MakePostForEO \\
\qquad\qquad (piseq, dfdsig, intm, maxm) \mid piseq \in eos\})
\end{array}
$$

The function *MakePostForEO* generates a post-expression for a specific execution order. An application of the quoted post-condition of the first data transformer in the execution order is generated (by *MakeQuotedApply*) and then *MakePostForEO* is called recursively with the remainder of the data transformers. A collection of intermediate state values $intm'$ is constructed in each recursion step in order to use the correct intermediate state values in the construction of a quotation for an operation. All quotations are combined in a conjunction.

[9] The auxiliary function *ExecutionOrders* generates a set of 'possible execution orders'. An execution order is a sequence of process identifiers suggesting a valid order in which the data transformers in a DFD with topology *dfdtopo* can be executed.

$$
\begin{array}{l}
MakePostForEO : ProcId^{+} \times DFDSig \times IntM \times IntM \to Expr \\
MakePostForEO\ (piseq, dfdsig, intm, maxm) \triangleq \\
\quad \mathsf{let}\ nid \quad = \mathsf{hd}\ piseq, \\
\quad\quad intm' = \{stid \mapsto \mathsf{if}\ mk\text{-}(stid, \text{READWRITE}) \in \\
\quad\quad\quad\quad\quad CollectStIds\ (\{dfdsig\ (nid)\}) \\
\quad\quad\quad\quad \mathsf{then}\ intm\ (stid) + 1 \\
\quad\quad\quad\quad \mathsf{else}\ intm\ (stid) \mid stid \in \mathsf{dom}\ intm\}, \\
\quad\quad pre' \quad = MakeQuotedApply\ (nid, dfdsig\ (nid), intm', \\
\quad\quad\quad\quad\quad\quad\quad\quad maxm, \text{POST}, \text{PRE}), \\
\quad\quad post' \ = MakeQuotedApply\ (nid, dfdsig\ (nid), intm', \\
\quad\quad\quad\quad\quad\quad\quad\quad maxm, \text{POST}, \text{POST})\ \mathsf{in} \\
\quad \mathsf{if}\ \mathsf{len}\ piseq = 1 \\
\quad \mathsf{then}\ mk\text{-}BinaryExpr\ (pre', \text{AND}, post') \\
\quad \mathsf{else}\ \mathsf{let}\ pred = mk\text{-}BinaryExpr\ (pre', \text{AND}, post')\ \mathsf{in} \\
\quad\quad mk\text{-}BinaryExpr\ (pred, \text{AND}, MakePostForEO\ (\mathsf{tl}\ piseq, dfdsig, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad intm', maxm))
\end{array}
$$

MakeQuotedApply generates the application of the quotation of a pre- or a post-condition of an operation. The configuration function *StateVarIntConf* receives information about where it is quoted from; the necessity for this was shown in example 2. Notice how *MakeQuotedApply* uses the rules for quoting (as explained above) through some auxiliary functions that extract the different kinds of arguments.

$$
\begin{array}{l}
MakeQuotedApply : \\
\quad (DFDId \mid MSId) \times Signature \times IntM \times IntM \times \\
\quad (\text{PRE} \mid \text{POST}) \times (\text{PRE} \mid \text{POST}) \to Apply \\
MakeQuotedApply\ (id, mk\text{-}(in', out, st'), intm, maxm, c, q) \triangleq \\
\quad \mathsf{let}\ inarg \quad = [FlowIdVarConf\ (in'\ (i)) \mid i \in \mathsf{inds}\ in'], \\
\quad\quad oldstarg = [\mathsf{let}\ mk\text{-}(s, m) = st'\ (i)\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{if}\ m = \text{READ} \\
\quad\quad\quad\quad \mathsf{then}\ StateVarIntConf\ (s, intm\ (s), maxm\ (s), c) \\
\quad\quad\quad\quad \mathsf{else}\ StateVarIntConf\ (s, intm\ (s) - 1, maxm\ (s), c) \mid \\
\quad\quad\quad\quad\quad i \in \mathsf{inds}\ st'], \\
\quad\quad outarg \ = [FlowIdVarConf\ (out\ (i)) \mid i \in \mathsf{inds}\ out], \\
\quad\quad starg \quad = [\mathsf{let}\ mk\text{-}(s, \text{-}) = st'\ (i)\ \mathsf{in} \\
\quad\quad\quad\quad StateVarIntConf\ (s, intm\ (s), maxm\ (s), c) \\
\quad\quad\quad\quad \mid i \in \mathsf{inds}\ st' \cdot \mathsf{let}\ mk\text{-}(\text{-}, m) = st'\ (i)\ \mathsf{in} \\
\quad\quad\quad\quad\quad\quad m = \text{READWRITE}]\ \mathsf{in} \\
\quad \mathsf{if}\ q = \text{PRE} \\
\quad \mathsf{then}\ mk\text{-}Apply\ (\text{"pre\_"} \curvearrowright OpIdConf\ (id), inarg \curvearrowright oldstarg) \\
\quad \mathsf{else}\ mk\text{-}Apply\ (\text{"post\_"} \curvearrowright OpIdConf\ (id), inarg \curvearrowright oldstarg \curvearrowright \\
\quad\quad\quad\quad outarg \curvearrowright starg) \\
\mathsf{pre}\ \ \forall\ mk\text{-}(s, m) \in \mathsf{elems}\ st' \cdot \\
\quad\quad s \in \mathsf{dom}\ intm \wedge s \in \mathsf{dom}\ maxm \wedge \\
\quad\quad m = \text{READWRITE} \ \Rightarrow \ intm\ (s) > 0
\end{array}
$$

This completes the presentation of the generation of implicit operations. Notice that it is possible to mechanize the process such that any DFD can be trans-

formed into one implicit VDM-SL operation definition capturing the semantics of the DFD.

4.2.3. Generation of Explicit Operations

The explicit definitions of operations for composing data transformers in a DFD are constructed following the same dependency strategy that is used for generating the implicit definitions. The principle for combining data transformers uses the same dependency information from the DFD. However, since the state of the DFD is not explicitly mentioned in the call of an operation, there is no problem with intermediate state values for the explicit definitions. Thus, the explicit definitions in general are shorter and easier to read than the implicit ones. Partitioning is dealt with by using the non-deterministic statement;[10] in this way the choice of execution order is left open.

Example 4

Before presenting the formal description of how DFDs as a whole can be transformed into explicit operation definitions, we show how the DFDs from the first three examples can be described explicitly.

The first DFD from figure 3 can be specified by the following explicit operation definition:

$$PQ : A \xrightarrow{o} C$$
$$PQ\,(a) \triangleq$$
$$\quad \mathsf{def}\ b = P(a)\ \mathsf{in}$$
$$\quad \mathsf{def}\ c = Q(b)\ \mathsf{in}$$
$$\quad \mathsf{return}\ c$$

Def-statements[11] are used to introduce the (intermediate) data flows.

For the DFD in figure 4 the following explicit operation can be generated:

$$PQ_{DS} : A \xrightarrow{o} C$$
$$PQ_{DS}\,(a) \triangleq$$
$$\quad \mathsf{def}\ b = P(a)\ \mathsf{in}$$
$$\quad \mathsf{def}\ c = Q(b)\ \mathsf{in}$$
$$\quad \mathsf{return}\ c$$

This operation is equivalent to the one generated for the DFD in example 3, because the state components that are modified by the different operation need not be explicitly mentioned in the call of these operations. In this respect, explicit operations in VDM-SL are very much similar to procedures in imperative programming languages accessing global variables.

The following explicit operation can be generated for the DFD in figure 5:

[10] A non-deterministic statement takes a set of statements and executes each of them in a non-deterministic order.

[11] A def-statement corresponds to a let-statement (or let-expression) except that it is legal at the right-hand-side of the equal sign to use an operation call that may modify the state.

$$PQR_{DS} : A \times D \xrightarrow{o} C \times E$$

$$\begin{array}{l} PQR_{DS}\,(a,d) \triangleq \\ \quad \| \ ((\mathsf{def}\ b = P(a)\ \mathsf{in} \\ \qquad \mathsf{def}\ c = Q(b)\ \mathsf{in} \\ \qquad \mathsf{def}\ e = R(d)\ \mathsf{in} \\ \qquad \mathsf{return}\ mk\text{-}(c,e)), \\ \qquad (\mathsf{def}\ e = R(d)\ \mathsf{in} \\ \qquad \mathsf{def}\ b = P(a)\ \mathsf{in} \\ \qquad \mathsf{def}\ c = Q(b)\ \mathsf{in} \\ \qquad \mathsf{return}\ mk\text{-}(c,e)), \\ \qquad (\mathsf{def}\ b = P(a)\ \mathsf{in} \\ \qquad \mathsf{def}\ e = R(d)\ \mathsf{in} \\ \qquad \mathsf{def}\ c = Q(b)\ \mathsf{in} \\ \qquad \mathsf{return}\ mk\text{-}(c,e))) \end{array}$$

The three different execution orders are incorporated in a non-deterministic statement. It is necessary to use a return statement at the end of each sequence statement in the nondeterministic statement (each represents a possible execution order) to ensure that a correct return value is created.
□

The function used to create operations for DFDs in the explicit style is called *MakeDFDExplOp*. The strategy is somewhat similar to the one that has been used for the implicit style. Here we also have a number of possible execution orders that must be taken into account. The type of the operation and the parameters to it are created through some auxiliary functions. The main part of the work is done by *MakeStmtForEO* that creates a statement for each possible execution order. If there is more than one possible execution order, then the final body is derived by making a non-deterministic statement of these statement bodies.

$$MakeDFDExplOp : DFDId \times DFDTopo \times DFDSig \rightarrow ExplOp$$

$$\begin{array}{l} MakeDFDExplOp\,(dfdid, dfdtopo, dfdsig) \triangleq \\ \quad \mathsf{let}\ mk\text{-}(in', out, st') = dfdsig\,(dfdid), \\ \qquad eos \quad = ExecutionOrders\,(dfdtopo), \\ \qquad optype \quad = MakeOpType\,(dfdsig\,(dfdid)), \\ \qquad parms \quad = [mk\text{-}PatternId\,(FlowIdVarConf\,(in'\,(i))) \mid \\ \qquad\qquad i \in \mathsf{inds}\ in'], \\ \qquad bodys \quad = \{MakeStmtForEO\,(piseq, dfdid, dfdsig) \mid \\ \qquad\qquad piseq \in eos\}, \\ \qquad body \quad = MakeNonDetStmt\,(bodys)\ \mathsf{in} \\ \quad mk\text{-}ExplOp\,(OpIdConf\,(dfdid), optype, parms, body) \end{array}$$

The function *MakeStmtForEO* is defined recursively. In each recursion step one data transformer is processed until all data transformers in the given partition have been incorporated. The strategy is the same as for *MakePostExpr* where a new (independent) data transformer is chosen. The function *MakeCallAndPat* creates a call of the operation for the given data transformer and the corresponding pattern against which the call must be matched. If the operation returns a value (i.e. $kind = \text{OPRES}$), the call is used in a def-statement. Otherwise it is a call statement which must be included as a part of a sequence of statements.

```
MakeStmtForEO : ProcId+ × DFDId × DFDSig → Stmt
MakeStmtForEO (piseq, dfdid, dfdsig) △
  let nid             = hd piseq,
      mk-(call, pat)  = MakeCallAndPat (nid, dfdsig (nid)),
      kind            = FindKind (dfdsig (nid)) in
  if len piseq = 1
  then let mk-(-, out, -) = dfdsig (dfdid),
           return'        = mk-Return (MakeResult (out)) in
       if kind = OPRES
       then mk-DefStmt (pat, call, return')
       else mk-Sequence ([call, return'])
  else let rest = MakeStmtForEO (tl piseq, dfdid, dfdsig) in
       if kind = OPRES
       then mk-DefStmt (pat, call, rest)
       else if is-Sequence (rest)
            then let mk-Sequence (sl) = rest in
                 mk-Sequence ([call] ⌢ sl)
            else mk-Sequence ([call, rest])
pre hd piseq ∈ dom dfdsig
```

This completes the presentation of the generation of explicit operations. Notice that here as well it is possible to mechanize the process such that any DFD can be transformed into one explicit VDM-SL operation definition that captures the semantics of the DFD. It is also worth noticing here that when the explicit specification style is used intermediate values are not visible, but that different execution orders influence the resulting operations.

5. Conclusions

In this paper we have defined a semantics for DFDs by formally specifying a transformation from DFDs to VDM specifications. In this section we give a brief overview of related work in the area of defining semantics for DFDs, and we conclude with some observations on our work and some ideas for further research.

5.1. Related work

When DFDs were originally introduced, they were presented as a graphical notation. The intended semantics of this notation was defined verbally, but the need for a formal base is now more commonly recognized, see e.g. [tHvdW92]. Work has been done on formalizing DFDs, with the intention of either disambiguating their meaning, or of using the formal semantics as a base for a combined formal/structured method.

In [Ran90] a translation back and forth between DFDs and Z specifications is described. [Ala88] contains an explanation of how DFDs can manually be transformed into an object-oriented design. The paper touches upon some problematic issues arising in a transformation from DFDs. In [SA] a small example of how a DFD can be transformed in Z is presented. However, no formal semantics of the DFDs is presented and it is not clear to what extent the transformation can be automated. In [BvdW89] some guidelines for how semantics can be attached to

DFDs are given. It is sketched how DFDs can be transformed into a Petri net variant combined with path expressions. In [ELP93] a complete semantics is provided for the Ward and Mellor version of SA/RT by means of high-level timed Petri nets. Here an executable subset of VDM-SL is also used to describe the mini-specifications of an SA/RT model. In [Adl88] a semantic base for guiding the decomposition process in the construction of a hierarchy of DFDs is presented. This work is based on graph theory in an algebraic setting. Kevin Jones uses VDM to provide a denotational style semantics of a non-conventional machine architecture (The Manchester DataFlow Machine) based on data flow graphs [Jon87]. In [FKV91] a rule-based approach for transforming SA products into VDM specifications is presented. Their VDM specifications are very explicit and hard to read, mainly because of the way decision tables have been taken into account. Polack et al. concentrate on the methodological aspects of combining SA notations and Z specifications [Pol92], the resulting combination is known as SAZ. Tse and Pong use extended Petri nets for formalizing DFDs [TP89]. France discusses an algebraic approach to modeling control-extended DFDs in [Fra92]. In [SFD92] an overview of several approaches to combining SA techniques and notations with formal methods (including our approach) is given.

The main result of the work presented in this paper with respect to other work in this area is that we have been able to capture the semantics of a DFD as a whole in a compositional way at a high level of abstraction, taking into account the whole hierarchy of DFDs that is created during an SA development, which to our knowledge has not been done before.

5.2. Status and Perspectives

With respect to the semantics of DFDs in terms of a formal transformation to VDM specifications the following observations can be made:

- An unambiguous interpretation of DFDs is available, which – due to the particular transformation chosen – is abstract. Consequently, there are few restrictions on the further development of the DFD into a software design.
- The transformation is executable, which opens up possibilities for automatically generating VDM specifications from DFDs. In this way, the initial effort needed to produce a formal specification is significantly decreased.
- The DFDs and their VDM counterparts can be regarded as equivalent views on the system, using different representations.

A few restrictions apply to our transformation, however. One of these is the exclusion of concurrent systems, whereas some SA extensions provide facilities for specifying such systems. We briefly mentioned how some of the DFD constructs would be interpreted if we had taken concurrency into account. A transformation from a real-time SA variant to a combination of VDM and e.g. CCS [Mil80], CSP [Hoa85] or Petri nets [Pet77] would be an interesting area for future research. We foresee that the main problem in automatically providing a concurrent specification description would be that such a description would have a very low level of abstraction. Intuitively it would be expected that each data transformer is transformed into a *process* and that all these processes are executed in parallel. This would result in a large number of processes due to the number of data transformers usually present in a DFD.

Concerns might also arise with respect to the size of the class of DFDs having no cyclic internal data flows and obeying the one-to-one mapping from input values to output values. In our experience, cyclic data flows are often used to model error situations which could also have been modeled by means of state components in data stores. Therefore, most DFDs with such cyclic structures can be rewritten using only acyclic structures, and therefore we believe that this restriction is not very important. With respect to the restriction to one-to-one mappings between input values and output values, we can say that usually the need for other mappings only occurs when DFDs are used as a design notation, but not when they are used as an (abstract) specification notation. Therefore, this restriction cannot be considered very important in our situation.

Acknowledgements

We would like to thank Michael Andersen, Hanne Christensen and John Dawes for their comments on this paper. We would also like to thank the anonymous referees of this paper for their useful suggestions for improvement. Peter Gorm Larsen acknowledges the financial support of the Commission of the European Communities under the COMETT programme (90/5199-Bc) and the Danish COWI-foundation.

References

[—92] ——. VDM Specification Language: Proto-Standard (Draft). Document N-246 (I-9), BSI IST/5/-/19 and ISO/IEC JTC1/SC22/WG19, December 1992.

[Adl88] Mike Adler. An Algebra for Data Flow Diagram Process Decomposition. *IEEE Transactions on Software Engineering*, SE-14(2):169–183, February 1988.

[AI91] Derek Andrews and Darell Ince. *Practical Formal Methods with VDM*. McGraw Hill, 1991.

[Ala88] Bruno Alabiso. Transformation of Data Flow Analysis Models to Object Oriented Design. In *OOPSLA'88 Proceedings*, pages 335–353. ACM, November 1988.

[Bea88] Stephen Bear. Structuring for the VDM Specification Language. In R. Jones R. Bloomfield, L. Marshall, editor, *VDM'88; VDM – The Way Ahead*, pages 2–25. Springer-Verlag, March 1988. LNCS 328.

[BJ82] D. Bjørner and C.B. Jones. *Formal Specification & Software Development*. Prentice Hall International, 1982.

[BvdW89] P.D. Bruza and Th. P. van der Weide. The Semantics of Data Flow Diagrams. Technical Report 89-16, University of Nijmegen, The Netherlands, October 1989.

[CY79] L.L. Constantine and E. Yourdon. *Structured Design*. Prentice Hall International, 1979.

[Daw91] John Dawes. *The VDM-SL Reference Guide*. Pitman (London, UK), 1991.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press (New Jersey, USA), 1979.

[ELP93] René Elmstrøm, Raino Lintulampi, and Mauro Pezzé. Giving Semantics to SA/RT by Means of High Level Timed Petri Nets. *Real-Time Systems*, 5(2/3):249–272, May 1993.

[FKV91] M.D. Fraser, K. Kumar, and V.K. Vaishnavi. Informal and Formal Requirements Specification Languages: Bridging the Gap. *IEEE Transactions on Software Engineering*, SE-17(5):454–466, May 1991.

[Fra92] Robert B. France. Semantically Extended Data Flow Diagrams: A Formal Specification Tool. *IEEE Transactions on Software Engineering*, SE-18(4):329–346, April 1992.

[GS79] Chris Gane and Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice Hall International, 1979.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Jon87] K.D. Jones. A Formal Semantics for a Dataflow Machine. In C.B. Jones D. Bjørner, editor, *VDM'87; VDM – A Formal Method at Work*, pages 331–355. Springer-Verlag, March 1987. LNCS 252.

[Jon90] C.B. Jones. *Systematic Software Development using VDM (2nd edition).* Prentice Hall International, 1990.

[LAMB89] Peter Gorm Larsen, Michael Meincke Arentoft, Brian Monahan, and Stephen Bear. Towards a Formal Semantics of The BSI/VDM Specification Language. In Ritter, editor, *Information Processing 89*, pages 95–100. IFIP, North-Holland, August 1989.

[Las93] Poul Bøgh Lassen. IFAD VDM-SL Toolbox. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, page 681, Berlin Heidelberg, April 1993. Springer-Verlag.

[LN86] G. Longworth and D. Nicholls. *SSADM Manual.* NCC, December 1986.

[LvKP+93] Peter Gorm Larsen, Jan van Katwijk, Nico Plat, Kees Pronk, and Hans Toetenel. SVDM: An Integrated Combination of SA and VDM. In *Proc. of the Methods Integration Conference, Leeds, UK, September 1991 (to appear)*, 1993.

[Mil80] A.J.R.G. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. LNCS 92.

[Pet77] J.L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Pol92] Fiona Polack. Integrating Formal Notations and Systems Analysis: using Entity Relationship Diagrams. *IEE/BCS Software Engineering Journal*, 7(5):363–371, September 1992.

[PvKP91] Nico Plat, Jan van Katwijk, and Kees Pronk. A Case for Structured Analysis/Formal Design. In *VDM'91; Formal Software Development Methods*, pages 81–105. Springer-Verlag, 1991. LNCS 551.

[Ran90] G.P. Randell. Translating Data Flow Diagrams into Z (and Vice Versa). Technical Report 90019, Procurement Executive, Ministry of Defence, RSRE, Malvern, Worcestershire, UK, October 1990.

[SA] Lesley Semmens and Pat Allen. Using Entity Relationship Models as a basis for Z Specifications.

[SFD92] L.T. Semmens, R.B. France, and T.W.G. Docker. Integrated Strudtured Analysis and Formal Specification Techniques. *The Computer Journal*, 35(6):600–610, December 1992.

[tHvdW92] A.H.M. ter Hofstede and T.P. van der Weide. Formalization of Techniques: Chopping Down the Methodology Jungle. *Journal of Information and Software Technology*, 34(1):57–65, January 1992.

[TP89] T.H. Tse and L. Pong. Towards a Formal Foundation for DeMarco Data Flow Diagrams. *The Computer Journal*, 32(1):1–12, January 1989.

[WM85] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems.* Yourdon Press (New Jersey, USA), 1985.

[You75] E. Yourdon. *Techniques of Program Structure and Design.* Prentice Hall International, 1975.