Towards a Calculus for UML-RT Specifications

R. Grosu, M. Broy, B. Selic*, Gh. Stefănescu‡

Institut für Informatik, TU München, D-80290 München

*ObjecTime Limited, K2K 2E4 Kanata, Ontario

‡Faculty of Mathematics, University of Bucharest, RO-70109 Bucharest

email:grosu,broy@in.tum.de, bran@ObjecTime.com, ghstef@stoilow.imar.ro

Abstract

The unified modeling language (UML) developed under the coordination of the Object Management Group (OMG) is one of the most important standards for the specification and design of object-oriented systems. This standard is currently tuned for real-time applications in the form of a new proposal, UML for Real-Time (UML-RT), by Rational Software Corporation and ObjecTime Limited. Because of the importance of UML-RT we are investigating its formal foundation in a joint project between ObjecTime Limited, Technische Universität München and the University of Bucharest. In this paper we present part of this foundation, namely the theory of flow-graphs.

1 Introduction

The specification and design of an interactive system is a complex task that has to work out data, behavior, intercommunication, architecture and distribution aspects of the modeled system. Moreover the specification has to assure the successful communication between the customer and the software expert. In order to fulfill these requirements, an UML-RT [8] specification for an interactive system is a combined visual/textual specification, called a *capsule class,* which is built hierarchically as shown in Figure 1, left.

A capsule class has associated two visual specifications: a *structure* specification and a *behavior* specification. The structure specification gives the architecture of the capsule in terms of other capsules and *connectors (or duplex channels)* between capsules. The connectors are typed, i.e., they have associated *protocol classes* defining the messages allowed to flow along the connectors. The types of the messages, and therefore the protocols themselves, are defined in terms of *data classes* or directly in C++ or RPL[1]. The flow of the messages along the channels is controlled by the *run-time system.*

The behavior of a capsule is controlled by a state transition diagram. The *state variables* and the *functions* occurring in this diagram are also defined in terms of data-classes, C++ or RPL. Moreover, the detailed description of the *actions* associated to a transition are given in C++ or in RPL. Hence, the UML-RT visual specifications build on the top of a *sequential object-oriented language.* Special actions like sending a message or setting a timer are performed by calling the run-time system. Hence, UML-RT also builds upon a communication and synchronization model. Since a sender may always send a message this is an *asynchronous* communication model.

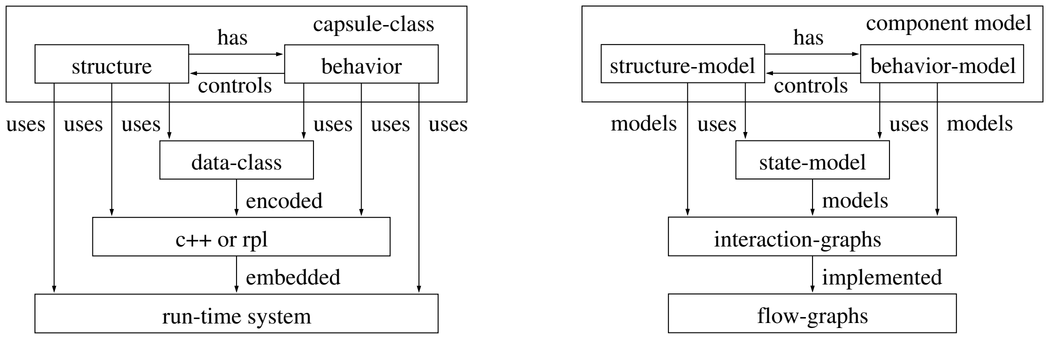

Figure 1: The layers of UML-RT

The semantics we currently define for UML-RT, in a joint project between Objec-Time Limited, Technische Universität München and the University of Bucharest, follows a similar hierarchy, as shown in Figure 1, right. It consists of a *structure-model*, a *behavior-model* and a *state-model.* Each model interprets an associated *interaction-graph.* These graphs closely resemble the UML-RT visual specifications. However, they are completely formalized and this makes them an ideal candidate for the semantics.

The structure-model defines the structure of a capsule class in terms of other capsules and connectors between these capsules. It also defines the synchronization between capsules. The behavior-model defines the behavior of a capsule in terms of hierarchical states and transitions between these states. The state-model is the equivalent of the data-classes and the object-oriented languages. It allows us to define arbitrary data-types and functions processing these types. Hence, it is used both by the structure and the behavior-models. In contrast to UML-RT, this is also a model for interaction-graphs. Hence, using our semantics, one can make UML-RT completely visual and independent from any particular programming language.

[1]RPL (Rapid Prototyping Language) is a very simple object-oriented language based on Smalltalk-80.

Finally, interaction graphs are implemented by *flow-graphs.* In contrast to interaction graphs which are appropriate for high-level design, the later ones are low-level graphs which make causality explicit.

Because of obvious space limitation, in this paper we concentrate on the theory of flow-graphs. This theory is characterized by three elements: a *visual notation,* a *textual notation* and a *calculus.* The visual notation consists of a set of graph-construction primitives presented in a visual form. They define the *user interface* to an abstract editor for diagrams. The textual notation consists of the same set of graph-construction primitives, presented in a textual form. They define an abstract *internal representation* of the above primitives. The textual form is *automatically generated* from the visual form, and it is usually hidden from the user. It can be roughly understood as the program that actually runs on the computer. Finally, the calculus is the engine that allows us to *transform* a diagram into another diagram that has the *same meaning* but optimizes time and/or space. Moreover it *determines* whether two diagrams are equivalent. The calculus consists of a set of equations which identify equal graphs. This immediately allows us to compare diagrams. Orienting the equations (e.g. from left to right) one obtains a *rewriting calculus,* i.e., an interpreter.

The rest of the paper is organized as follows. In Section 2 we introduce the syntax of flow-graphs. In Section 3 we present the calculus of flow-graphs. In Section 4 we present a non-trivial flow-graphs transformation by using the above calculus. Finally in Section 5 we draw some conclusions.

2 Flow-Graphs Syntax

A hierarchical flow-graph consists of a set of nodes connected by a set of directed arcs. In the following we treat the syntax of flow-graphs and their axioms.

2.1 Nodes

Consider the graph shown in Figure 2, left, consisting of two nodes A and B and an arrow m from A to B. Regarding A and B as sets (or types) m can be regarded as a function (or transition relation) mapping elements of A into elements of B. The corresponding textual notation is then $m : A \to B$.

Figure 2: Alternative representation of nodes and arrows

Since m turns out to be the most important element of the graph, one can change the roles of nodes and arrows and obtain the graph shown in Figure 2, right. This graph has a very intuitive computational interpretation: m is a computation unit receiving on the *input port* messages of type A from the *environment* and sending along the *output port* messages of type B to the environment. Hence, this notation is well suited for the description of *open systems.* These are systems that are connected to the environment. In the following, we shall use only this notation.

For example, the right-hand-side notation of Figure 2 is used in UML-RT both in the architecture and in the behavior specifications, as shown in Figure 3.

Figure 3: Actors and states in UML-RT

In the *architecture specification*, m is a *capsule* receiving messages of type A and sending messages of type B along a *duplex port.* The tuple (A^-, B^+) is said to be the *protocol* defining the *input* and the *output* messages flowing along the port. In the *behavior specification*, m is an *extended state* with an *entry point* A of an incoming transition and an *exit point* B of an outgoing transition.

2.2 Graph Primitives

As shown in Figure 4, we consider three operators on nodes: *sequential composition*, *visual attachment* and *feedback*.

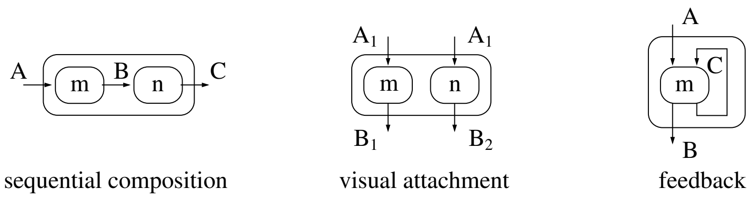

Figure 4: The node operators

Sequential composition. The most basic way to *connect* two nodes is by *sequential composition,* i.e., as shown in Figure 4, left, by connecting the output of one node to the input of the other node, if they have the same type. Textually we denote this operator by the semicolon ;. Given $m : A \to B$ and $n : B \to C$ we define $m; n$ to be of type $m; n\ : A \to C$. Regarding the nodes as computation units, Figure 4, left, says that the output produced by m is directed to the input of n. The connection between m and n as well as the units m and n themselves, are internal to $m; n$. In other words, $m; n$ does not only define a *connection* relation but also a *containment* relation.

Visual attachment. By *visual attachment* we mean that nodes and corresponding arrows are put one near the other, as shown in Figure 4, middle. To obtain a textual representation for visual attachment, we need therefore an attachment operator both on arrows and on nodes. We denote this operator by $*$. Given two arrows A and B their visual attachment is expressed by $A * B$. Given two nodes $m : A_1 \to B_1$ and $n : A_2 \to B_2$ their visual attachment is expressed as $m * n : A_1 * A_2 \to B_1 * B_2$. Visual attachments also defines a containment relation. We say that m and n are contained in $m * n$.

Feedback. Sequential composition allows to connect nodes in a linear way. To deal with loops we introduce a *feedback operator* that, as shown in Figure 4,right, allows to connect the rightmost output of a node to the rightmost input of the same node, if they have the same type. Given $m : A * C \to B * C$ we define $m \uparrow^{C}_{A,B}: A \to B$. Feedback also introduces a containment relationship. We say that m and the feedback arrow are contained in $m \uparrow^{C}_{A,B}$.

2.3 Connectors

The above graph primitives already allow us to construct a wide variety of graphs from a given set of nodes. Since the primitives are strongly typed, the arrows of the involved nodes have to match the arrows expected by the primitive. To take care of possible mismatches we introduce, as shown in Figure 5, four *standard nodes*: *identity* i_A, *transposition* ${}^{A}\mathsf{X}^{B}$, *identification* $\vee_A$ and *ramification* $\wedge^A$. Because they allow us to "plug in nodes" we call them *connectors.*

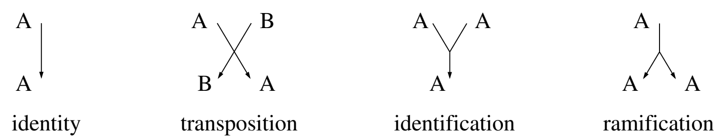

Figure 5: The connectors

Identity. In conjunction with visual attachment, the *identity* connector $i_A : A \to A$ allows us to adapt the interface of a node in a way that makes explicit what does not change.

Transposition. The *transposition* connector ${}^{A}\mathsf{X}^{B} : A*B \to B*A$ allows us to take care of the left/right mismatch, by exchanging the left arrow with the right arrow.

Identification The *identification* connectors $\top_A : E \to A$ and $\vee_A : A*A \to A$ allow us to *merge (or concentrate)* control flows in behavior specifications and to *synchronize* communication in architecture specifications.

Ramification. The *ramification* connectors $\perp^A : A \to E$ and $\wedge^A : A \to A * A$ allow us to *distribute* control in behavior specifications and to *copy (or broadcast)* messages in the architecture specifications.

3 Flow-Graphs Properties

In this section we present the properties of the above graph-construction primitives and connectors. They define the calculus of flow-graphs.

3.1 General Flow-Graphs

In this section we deal with general flow-graphs. In the following section we treat specific aspects of data-flow and control-flow graphs.

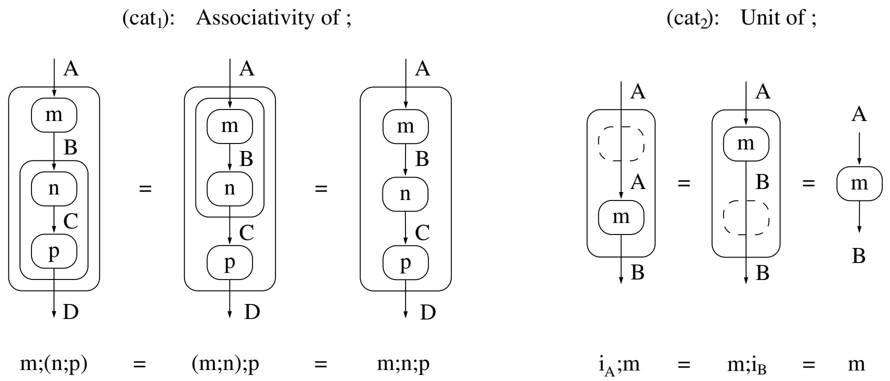

Figure 6: Properties of sequential composition

3.1.1 Sequential Composition

This section defines the axioms for sequential composition.

Associativity and unit. Suppose we connect three nodes, m, n and p as shown in Figure 6, left. Then we require that $m; (n; p)$ maps A to D in the same way as $(m; n); p$, i.e., sequential composition is *associative.* As a consequence, we can drop the internal box in the visual notation and the parenthesis in the textual notation. To put it in another way, an implementation using a left parsing of the composition is considered to be equivalent with one using a right parsing or even with one using a mixed parsing.

For each arrow A we also require the existence of an identity node i_A, i.e., of a node that forwards its input to the output. Visually we represent this node by an *arrow* or by an arrow surrounded by a *dashed box* if we want to emphasize it. This node plays the role of a *neutral element* for composition, as shown in Figure 6, right.

In mathematical terms, a graph equipped with a composition operation and identities such that composition is associative and has identities as neutral elements is called a *category* (cat).

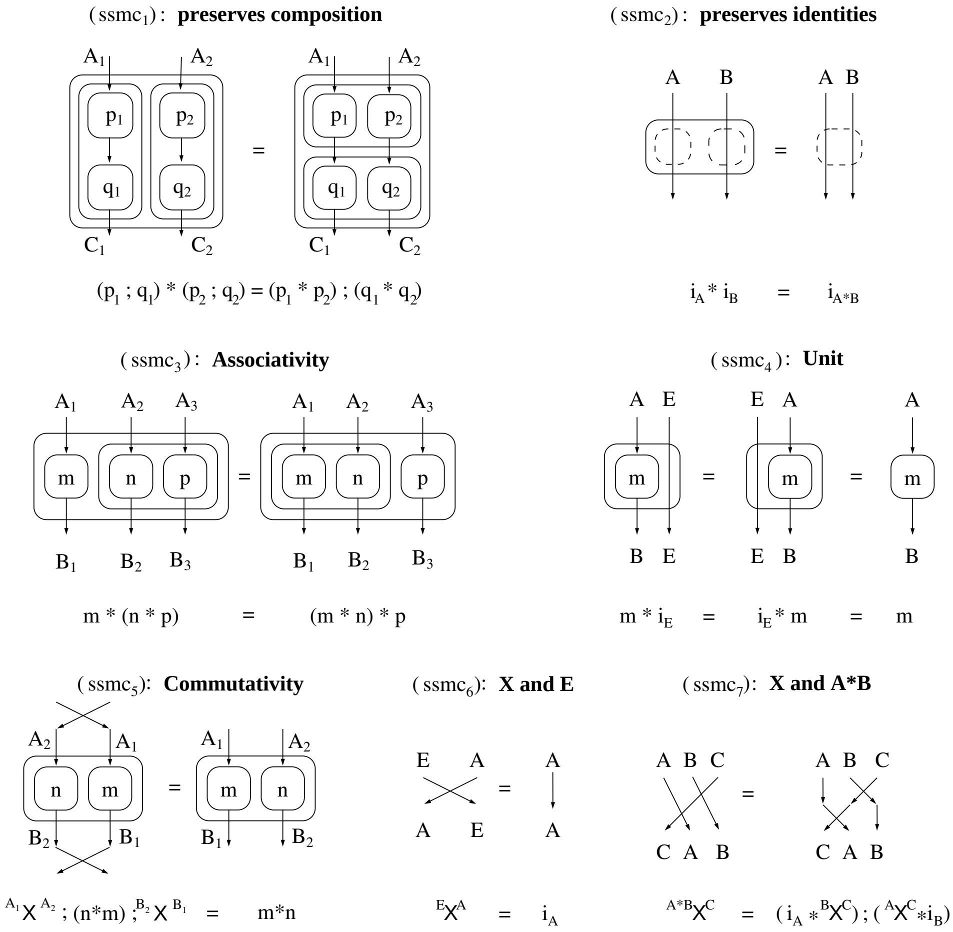

Figure 7: Properties of visual attachment

3.1.2 Visual Attachment

Functoriality. Since m and n can themselves be composed nodes of the form $m = p_1; q_1$ and $n = p_2; q_2$ one can wonder how visual attachment relates to sequential composition. A very reasonable assumption is that visual attachment preserves sequential composition and identities as shown by $(ssmc_{1-2})$ in Figure 7. In mathematical terminology, a mapping $*$ both on nodes and on arrows having the above properties is called a *functor.*

Associativity and unit. Similarly to sequential composition we require that the visual attachment of nodes is *associative* with *unit* the identity node i_E as shown by $(ssmc_{3-4})$ in Figure 7. The arrow E is the unit for the visual attachment of arrows. In mathematical terminology, a category equipped with a functor which is associative and has a neutral element is called a *strict monoidal category* (smc).

Commutativity. Two nodes m and n may be visually attached in two different ways: $m * n$, i.e., with m on the left or $n * m$, i.e., with n on the left. Since we are mainly interested in the "one near the other" relation, these two attachments should be equivalent. However, one can in general not assume that $A * B = B * A$ (think about sets product), but one can safely assume that $A * B$ is isomorphic to $B * A$. Denote by ${}^{A}\mathsf{X}^{B} : A * B \to B * A$ the corresponding *transposition* isomorphism. Then commutativity is expressed by ($ssmc_5$) in Figure 7. Transposition is extended to the neutral arrow E and to composed arrows $A * B$ as shown by ($ssmc_{6-7}$). In mathematical terminology, X makes the strict monoidal category *symmetric* (ssmc).

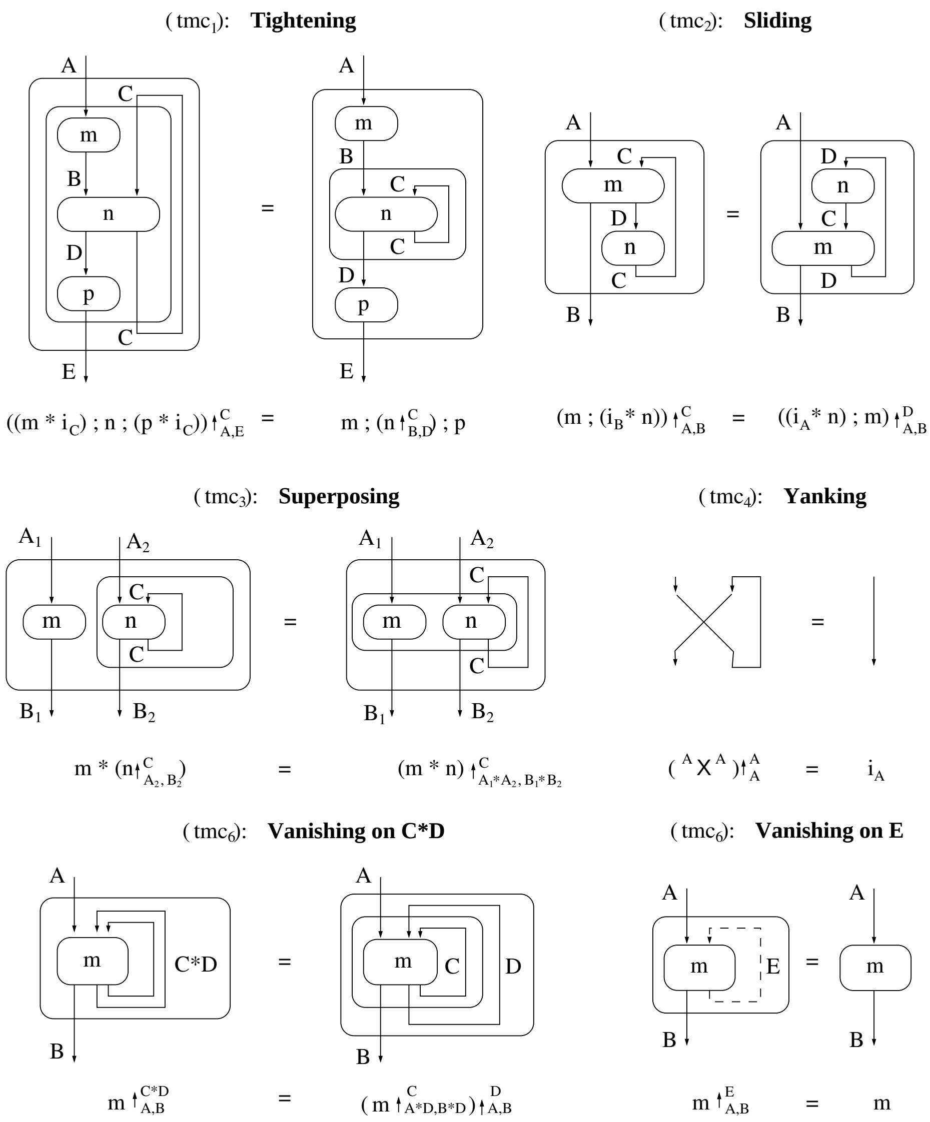

Figure 8: Properties of feedback

3.1.3 Feedback

Naturality properties. The desired relation of feedback to sequential composition is shown by (tmc_{1-2}) in Figure 8. The first equation is called *tightening*, because it tightens the scope of the feedback. The second equation is called *sliding*, because

n slides along the feedback loop. In mathematical terminology one says that the feedback operator $\uparrow^C_{A,B}$ is *natural* in the arrows A, B and C.

Superposing and yanking. The relation of the feedback to visual attachment and to transposition is shown by (tmc_{3-4}) in Figure 8. The first equation is also called *superposing*, because it superposes feedback over visual attachment. The second equation is also called *yanking,* because it behaves like yanking a rope.

Feedback on composed arrows. Since the category is monoidal, it also contains arrows E and $A * B$. As a consequence, we have to define the behavior of feedback on these arrows, too. We do this by defining the behavior on E and by reducing the behavior on $A * B$ to the behavior on A and on B, as shown by (tmc_{5-6}) in Figure 8. These equations are also called the *vanishing* equations because they show how to decompose the feedback loop.

In mathematical terminology, a strict symmetric monoidal category together with a feedback operator satisfying the tightening, sliding, superposing, yanking and vanishing axioms is called a *trace monoidal category* (tmc).

3.1.4 Identification

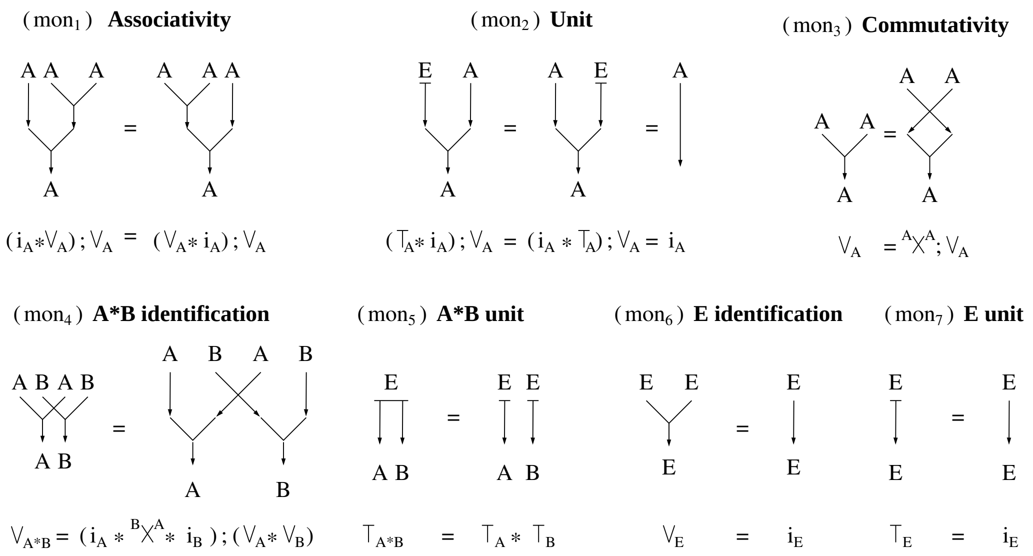

Figure 9: Properties of identification

Associativity. Since visual attachment is associative (with unit i_E) and it commutes with transposition, we require as shown by (mon_{1-3}) in Figure 9, that $\vee_A : A{*}A{\to}A$ is also associative (with unit $\top_A : E{\to}A$) and that it commutes with transposition.

The equation (mon_1) allows to construct (or replace) an n-ary identification either by a left or by a right parse of binary identifications. The equation (mon_2) allows to eliminate identifications and the equation (mon_3) allows to eliminate transpositions. All three equations may be used to optimize the internal representation. In mathematical terminology, one says that the arrow A equipped with the commutative and associative connector $\vee_A$ with neutral element $\top_A$ is a *commutative monoid.*

Extension to arbitrary arrows. The identification connectors are extended to E and $A * B$ as shown by (mon_{4-7}) in Figure 9. The equations (mon_{6-7}) allow to eliminate identifications of empty arrows. The equations (mon_{4-5}) give an inductive algorithm for the construction of the identification connectors on composed arrows. Hence, each arrow A in the monoidal category has a *commutative monoid structure.*

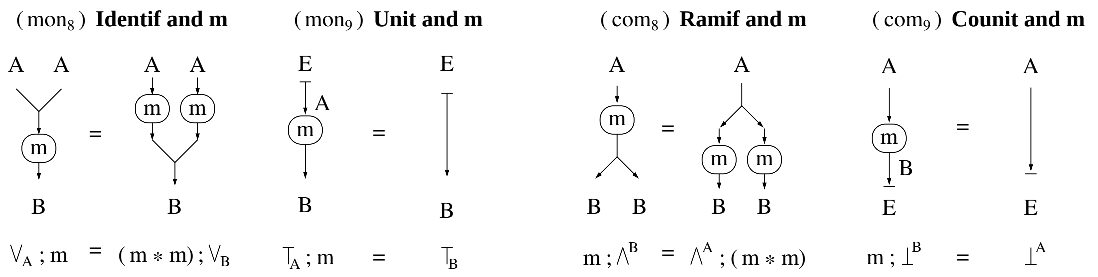

Figure 10: Properties of morphisms

Monoid morphisms. Nodes that preserve this structure are called *monoid morphisms.* Their morphism property is given by (mon_{8-9}) in Figure 10. The morphism property has a tremendous role in *optimization* because it allows to eliminate nodes. An example for such an optimization is given in Section 4.

3.1.5 Ramification

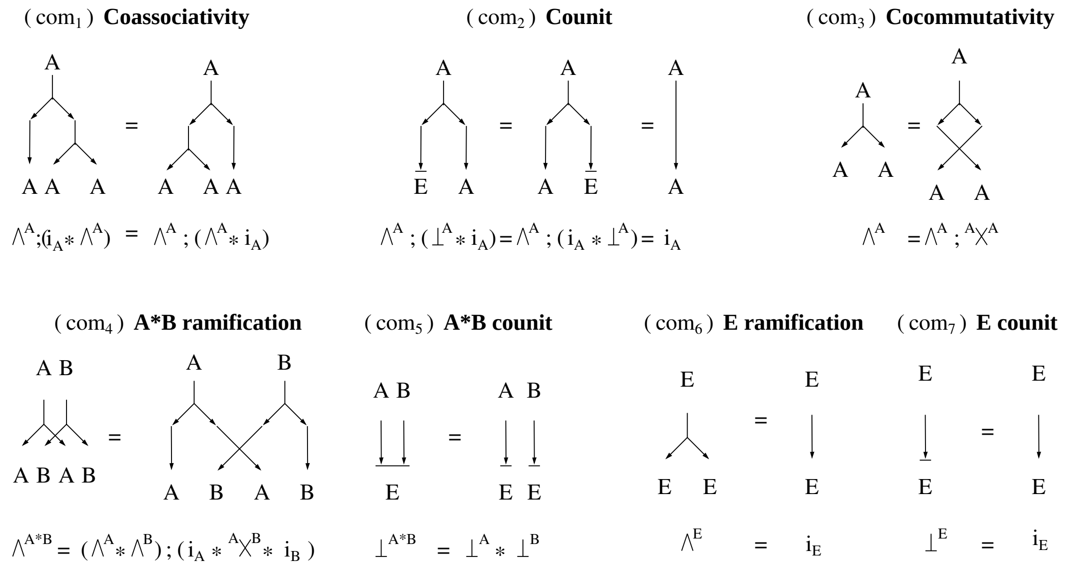

Figure 11: Properties of ramification

Coasociativity and coneutral. The coassociativity and cocommutativity properties of the ramification are the mirror image of the corresponding properties of identification, as shown in Figure 11. As with monoids, (com_1) allows to optimize sequences of binary ramifications, (com_2) allows to eliminate ramifications and (com_3) allows to eliminate transpositions. In mathematical terminology, one says that the arrow A equipped with the cocommutative and coassociative connector $\wedge^A$ with neutral element $\perp^A$ is a *commutative comonoid.*

Extension to arbitrary arrows. The ramification connectors are extended to $A * B$ and E as shown by (com_{4-7}) in Figure 11. Equations (com_{6-7}) allow to eliminate ramifications of empty arrows and equations (com_{4-5}) give an inductive algorithm for the construction of ramification on composed nodes. Hence, each arrow A in the monoidal category also has a *cocommutative comonoid structure.*

Comonoid morphisms. Nodes that preserve this structure are called *comonoid morphisms.* Their morphism property is given by (com_{8-9}) in Figure 10. The morphism property plays a very important role in *optimization* because it allows to eliminate nodes.

3.1.6 Identification and Ramification

Since visual attachment together with identification and ramification generates both a monoid and a comonoid structure on each arrow, one has to consider how these structures relate to each other. A very reasonable requirement is that identification preserves the structure generated by ramification or equivalently that ramification preserves the structure generated by identification. In other words, identification is a comonoid morphism and ramification is a monoid morphism. In analogy with bialgebras, we call A in this case a *bimonoid.* The requirement that $\vee_A$ is a comonoid morphism is given by (bim_{1-2}) in Figure 12.

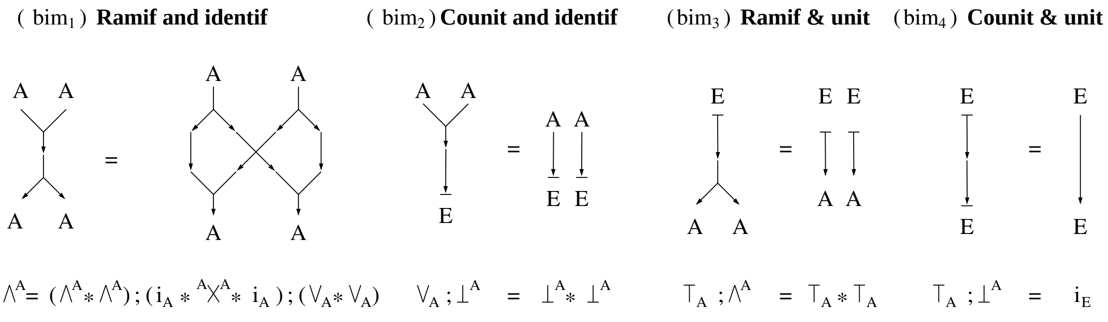

Figure 12: Bimonoidal properties

From a practical point of view, the equations (bim_1) and (bim_2) allow to eliminate identifications and ramifications. The requirement that $\top_A$ is a comonoid morphism is given by (bim_{3-4}). Again, from an optimization point of view, (bim_{3-4}) allow to eliminate identifications and ramifications.

The flow-graphs equations presented so far express the properties common both to structure and behavior diagrams. In the next sections we introduce additional properties, which further refine flow-graphs. These properties are closely related to the monoid- and comonoid-morphism properties.

3.2 Data-Flow Graphs

A *data-flow graph* is a flow-graph such that for each arrow A there is a *unique* node mapping this arrow to the unit arrow E, namely the connector $\perp^A$. In mathematical terminology, one says that *each node* m satisfies the comonoid-morphism property (com_7) and that E is a *terminal element.* We write E in this case also as 1. In *deterministic* data-flow graphs any node m also satisfies the comonoid-morphism property (com_6), i.e., each node in a deterministic data-flow graph is a *comonoid morphism.*

The connector $\perp$ allows us to define two *projection connectors* π_1 and π_2 as shown in Figure 13. Using the comonoid properties one can show that for any $m : C \rightarrow A$ and $n : C \rightarrow B$, the projections have the properties given by $(prod_{1-2})$ in Figure 13.

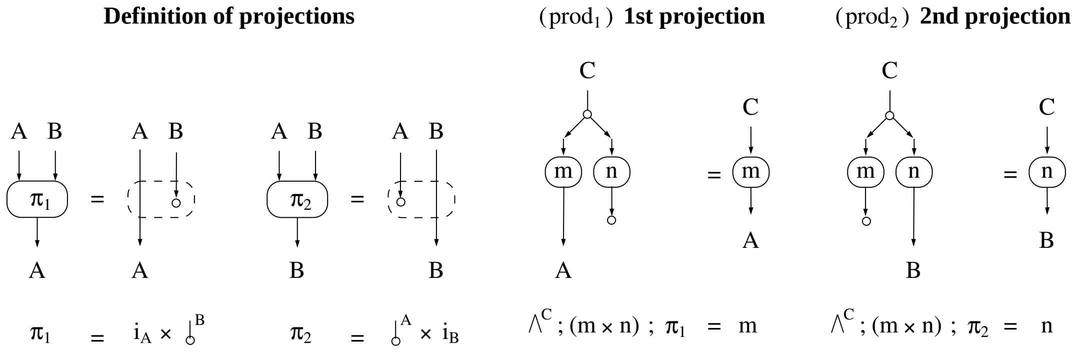

Figure 13: Projection connectors and their properties

The uniqueness of $\perp^A$ and $\perp^B$ assures that π_1 and π_2 are the *unique* nodes (up to isomorphism) having the properties $(prod_{1-2})$. The composed node $\wedge^A; (m * n)$ is also written as (m, n) and the above axioms as $(m, n); \pi_1 = m$ and $(m, n); \pi_2 = n$.

In mathematical terminology, $A * B$ together with the unique connectors π_1 and π_2 satisfying $(prod_1)$ and $(prod_2)$ is a *product* and data-flow graphs have finite products. The monoidal functor $*$ is written in this case as $\times$ and the identification and ramification connectors as $\curlyvee, \top$ and $\curlywedge, \perp$ respectively. The ramification connector is also called in this case the *diagonal connector.*

3.3 Control-Flow Graphs

A *control-flow graph* is a flow-graph such that for each arrow B there is a *unique* node mapping the unit arrow E to B, namely the connector $\top_B$. In mathematical terminology, one says that *each node* m satisfies the monoid-morphism property (mon_7) and that E is an *initial element*. We write E in this case also as 0. In *deterministic* control-flow graphs any node m also satisfies the monoid-morphism property (mon_6), i.e., each node in a deterministic control-flow graph is a *monoid morphism*.

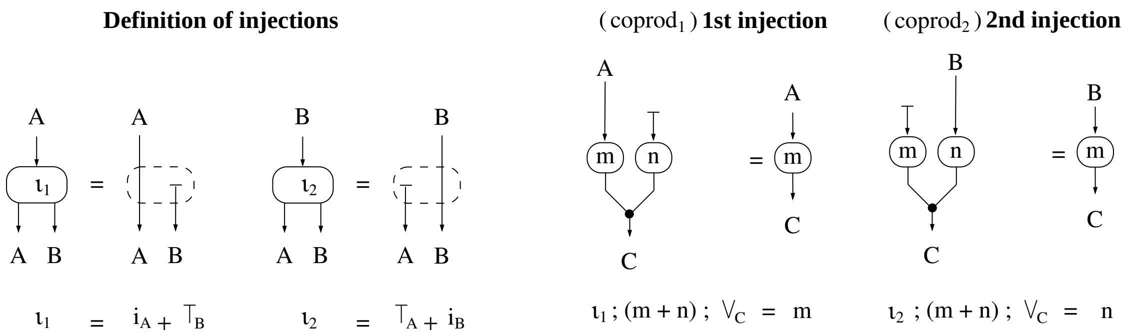

Figure 14: Injection connectors and their properties

The connector $\top$ allows us to define two *injection connectors* ι_1 and ι_2 as shown in Figure 14. Using the monoid properties one can show that for any $m : A{\to}C$ and $n : B{\to}C$, the injections have the properties given by the equations ($coprod_{1-2}$) in Figure 14.

The uniqueness of $\top_A$ and $\top_B$ assures that ι_1 and ι_2 are the *unique* nodes (up to isomorphism) having the properties ($coprod_{1-2}$). The composed node $(m * n); \vee_C$ is also written as $[m, n]$ and the above axioms as $\iota_1; [m, n] = m$ and $\iota_2; [m, n] = n$. The node $[m, n]$ works by case analysis: if its argument comes from A then it applies m; if it comes from B then it applies n.

In mathematical terminology, $A * B$ together with the unique connectors ι_1 and ι_2 satisfying ($coprod_{1-2}$) is a *coproduct (or sum)* and control-flow graphs have finite coproducts. The monoidal functor $*$ is written in this case as $+$ and the identification and ramification connectors as $\succ\!\!\bullet, \top$ and $\bullet\!\!\prec, \perp$ respectively. In this case the identification connector is also called the *codiagonal connector*. Note that control-flow graphs are obtained from data-flow graphs by *reversing* the direction of arrows. We therefore say that control-flow graphs are *dual* to data-flow graphs.

3.4 Mixed Flow-Graphs

A *mixed flow-graph* is a flow-graph having both *sums* and *products* such that the products distribute over sums, i.e., for any arrows A, B, C there exist an isomorphism $(A{\times}B){+}(A{\times}C) \cong A{\times}(B{+}C)$. This isomorphism is a *new connector* denoted by δ.

In mathematical terminology $\times$ and $+$ together with δ define a *distributive category* structure on flow-graphs. To explicitly distinguish sums from products in mixed flow-graphs, we use different shadings and interface layering, as shown in Figure 15. This information may also be left implicit.

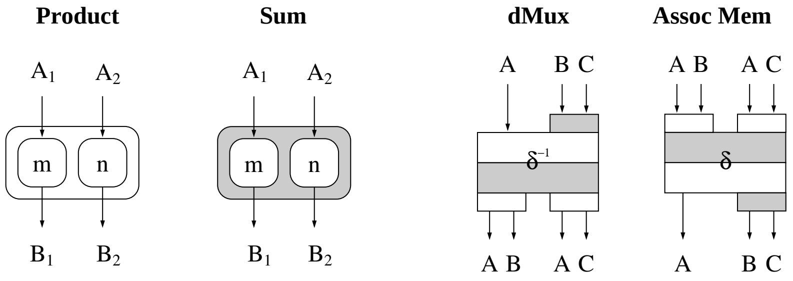

Figure 15: The shading conventions

The connectors δ^{-1} and δ may be understood as a generalization of *de-multiplexers* and *associative memories.* For the particular case that $B = C = 1$ we obtain $B{+}C = 1{+}1 = \mathbb{B}$ and $A{\times}1 = A$, where $\mathbb{B}$ is the set of Booleans. In this case the connector $\delta^{-1} : A{\times}\mathbb{B}{\to}A{+}A$ acts as a de-multiplexer. Given a data input $a{\in}A$ and a control input $b{\in}\mathbb{B}$, it forwards a along the first output if $b = (*{,}0)$ and along the second output if $b = (*{,}1)$. The values $(*{,}0)$ and $(*{,}1)$ are often written simply as 0 and 1 or as **false** and **true**. The connector $\delta : A{+}A{\to}A{\times}\mathbb{B}$ acts as an associative memory. It forwards a on the data output and the channel number where a was received on the control output.

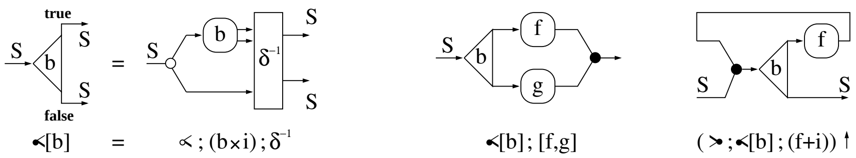

Figure 16: Branching, if and while

Given a predicate $b : S{\to}\mathbb{B}$, we can define with the help of δ^{-1} the *branching connector* of UM-RT-Charts as a *mixed flow-graph* as shown in Figure 16, left. This connector used in conjunction with sequential composition and identification allows to define the **if** b **then** f **else** g **fi** construct as in Figure 16, middle. It also allows to define in conjunction with feedback the **while** b **do** f **od** construct, as shown in Figure 16, right. Binary branching may be easily extended to n-ary branching by replacing $\mathbb{B}$ with $\mathbf{n} = n \cdot 1$.

Isomorphism properties. The properties (dis_{1-2}) in Figure 17 say that δ and δ^{-1} are inverse to each other.

Mixed superposing. An important question to answer is how does the *additive-feedback* relate to the *product* and dually, the *multiplicative-feedback* to the *sum.* The corresponding properties as given by (dis_{3-4}) in Figure 17.

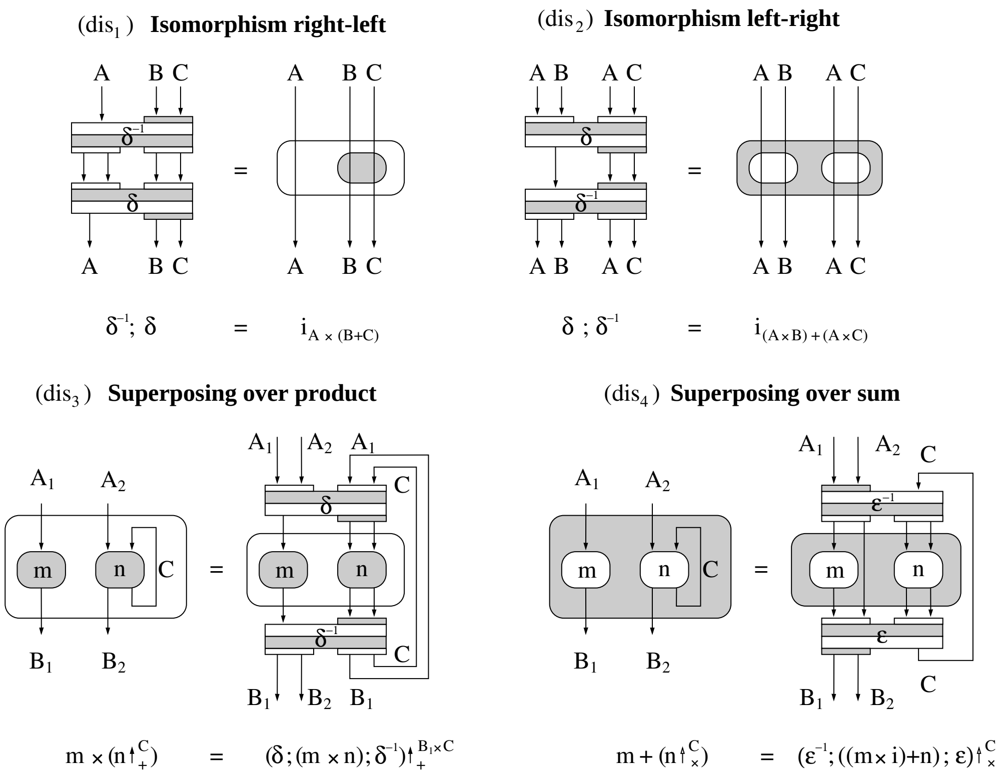

Figure 17: Properties of distributivity

Intuitively, (dis_3) says that m and n compute in parallel and output the result simultaneously. If m is faster then n then it has to idle until n finishes its computation. This is achieved by using *feedback* (hence $A_1 = B_1$) and by requiring that nodes are *idempotent*, i.e., that $m(m(a)) = m(a)$. From another perspective, (dis_3) allows to push feedback outside an expression.

Intuitively, (dis_4) says that (in each time unit) either m or n is allowed to compute the result, depending on the control information in the input data ($\epsilon = \mathsf{X}; \delta$). If the input is of the form $(a_1, 0)$ then the output is $(b_1, 0)$ and it is produced by m. In this case, the current feedback value c for n has to be preserved. This is achieved by putting m in parallel with the *identity* which simply forwards c to the output. If the input is of the form $(a_2, 1)$ then the output is of the form $(b_2, 1)$ and it is computed by n according to a_2 and the feedback input c. From another perspective, dis_4 allows to push feedback outside an expression.

4 Example of Flow-Graph Transformation

Suppose a first team has developed a component m, a second team used m to implement a component $p = (\vee; n; i{*}m) \uparrow$ and a third team used m and p to implement a component $q = m; p$. The question is, if this component may be optimized. This is definitely possible, as shown in Figure 18. First, we use sliding

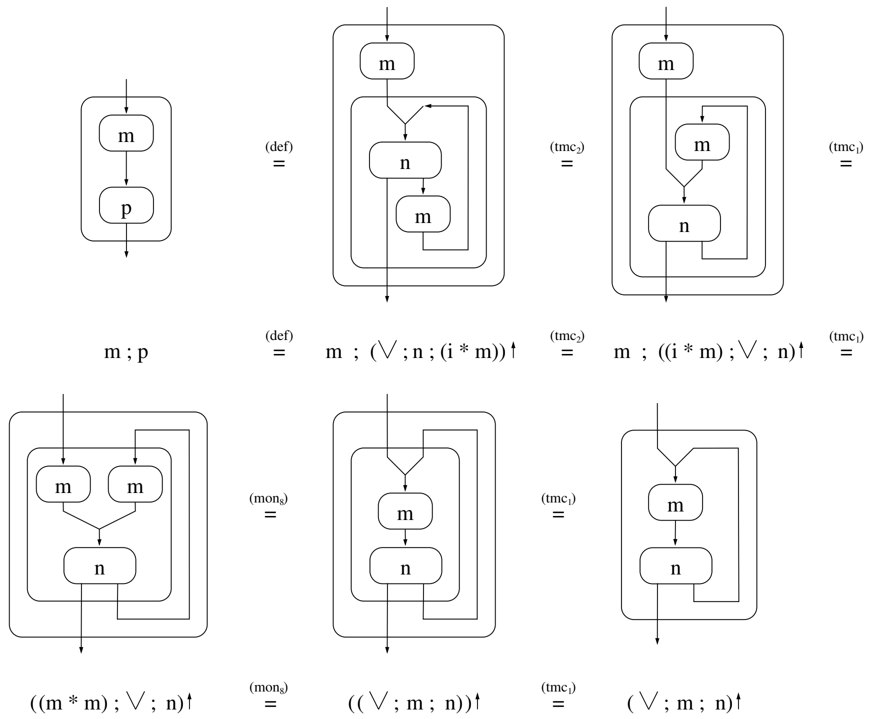

Figure 18: A graph transformation

(tmc_2) to move m above n. Then we use tightening (tmc_1) to move the first m inside the internal box. Then we use the morphism property (mon_8) to move m after the identification. Finally we use tightening again (tmc_1) to drop the internal box.

This transformation is non-trivial. Both the visual representation (the application-designer interface) and the textual representation (the tool-designer interface) are not immediately related to the original diagram and textual representation. Moreover, the final component was *optimized* because we eliminated one component m. This may have important consequences if m is a large (hardware) component.

5 Conclusions

The main benefits of the flow-graph theory can be summarized as follows. First, it introduces a set of graph-construction primitives in a *consistent way*. This diminishes the arbitrariness in the choice of these primitives. Second, it provides a mathematically *precise semantics* for these primitives. This is very useful as a *reference* both for tool designers and for system development engineers because it *eliminates misinterpretation.* Third, it provides a calculus which allows us to *compare* and to *optimize designs* and even to do *rapid prototyping.* While the visual notation is the interface to system engineers, the textual notation is the interfaces to tool developers.

The general calculus of flow-graphs is not new. It was already presented in the context of flow-charts in [4]. The axioms for feedback were independently rediscovered in [6]. The intuitive names for these axioms were borrowed from that paper. Similar properties for identification and ramification were also given in [2]. The flow-graphs operators were also used for the description of interactive systems in [3] and for dynamic systems in [5].

The treatment of data-flow and control-flow graphs as specialization of flow-graphs is to our knowledge new. The same holds for the treatment of mixed flow-graphs. In particular, for the axioms (bis_{3-4}) The first one relates additive (or timeless) feedback to multiplicative (or parallel) composition. The second one relates multiplicative (or timed) feedback to additive (or alternative) composition. Conditions related to (bis_3) also occur in [7]. However, this paper lacks the feedback operators, which makes the formalization more complicated and less transparent.

The use of the flow-graph theory in the context of visual formalisms, especially for UML-RT is also new. Moreover, this theory also forms the basis for the development of the theory of interaction-graphs. These are particularly relevant for UML-RT structure diagrams and for the definition and use of procedures in UML-RT behavior diagrams. By treating structure-diagrams and procedures as ordinary data which can be manipulated and sent along channels, interaction-graphs also allow the description of mobile systems. The axiomatization of interaction-graphs is very similar in spirit to the general form of the geometry of interaction [1].

References

[1] S. Abramski. Retracing some paths in process algebra. In *Seventh International Conference on Concurrency Theory (Concur'96), Lecture Notes Computer Science 1055*, pages 21–33, 1996.

[2] D.B. Benson. Bialgebras: Some foundations for distributed and concurrent computation. *Fundamenta Informaticae*, 12:427–486, 1989.

[3] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[4] V.E. Căzănescu and Gh. Stefănescu. Towards a new algebraic foundation of flowchart scheme theory. *Fundamenta Informaticae*, 13:171–210, 1990.

[5] R. Grosu and K. Stølen. A Model for Mobile Point-to-Point Data-flow Networks without Channel Sharing. In *Proc. of the 5th Int. Conf. on Algebraic Methodol-*

ogy and Software Technology, AMAST'96, Munich, pages 505–519. LNCS 1101, 1996.

[6] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.*, 119:447–468, 1996.

[7] N Sabadini, S. Vigna, and RFC Walters. A note on recursive functions. *Mathematical Structures in Computer Science*, 6:127–139, 1996.

[8] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Available under http://www.objectime.com/uml, April 1998.