Chapter 6

WHAT IS BEHIND UML-RT?

Radu Grosu
Institut für Informatik, TU München, D-80290 München
grosu@in.tum.de

Manfred Broy
Institut für Informatik, TU München, D-80290 München
broy@in.tum.de

Bran Selic
ObjecTime Limited, K2K 2E4 Kanata, Ontario
bran@ObjecTime.com

Gheorghe Stefănescu
Faculty of Mathematics, University of Bucharest, RO-70109 Bucharest
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. Our results clearly show that the visual notation of UML-RT is not only very intuitive but it also has a very deep mathematical foundation. In a previous paper (see [GBSS98]) we presented part of this foundation, namely the theory of flow graphs. In this paper we use flow graphs to define the more powerful theory of interaction 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 specification for an interactive system (see [SR98]) is a combined visual/textual specification, called a *capsule class*, which is built hierarchically as shown in Figure 6.1, 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 the protocols themselves, are defined in terms of *data classes* or directly in C++. The

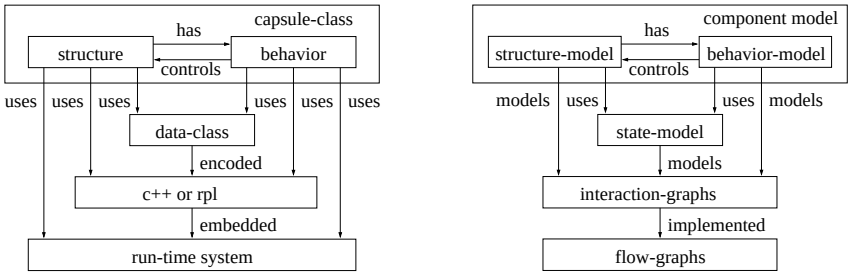

Figure 6.1 The layers of UML-RT and their semantic characterization

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 or in C++. Moreover, the detailed description of the *actions* associated to a transition are given in C++. Hence, the UML-RT visual specifications build on 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.

The semantics we currently define for UML-RT in a joint project between ObjecTime Limited, Technische Universität München and the University of Bucharest, follows a similar hierarchy, as shown in Figure 6.1, 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 of UML-RT.

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

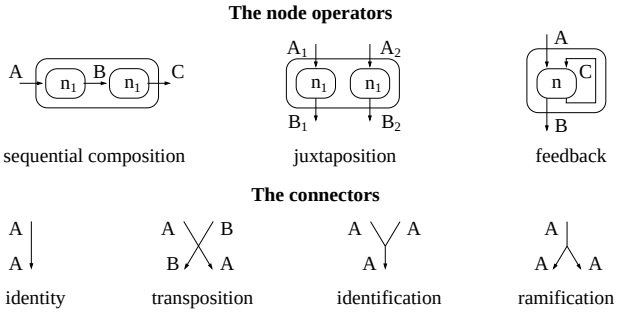

Figure 6.2 Flow graphs

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.

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.

In our earlier paper (see [GBSS98]) we presented the theory of flow graphs. In this paper we use flow graphs to define the theory of interaction 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 semantically 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 revise the basic elements of the theory of flow graphs. In Section 3 we first motivate the need for interaction graphs. Then we use the theory of flow graphs to define interaction graphs and their associated properties. Finally in Section 4 we draw some conclusions.

2. FLOW GRAPHS

Flow graphs (see [GBSS98]) are constructed by using, as shown in Figure 6.2, three operators on nodes – *sequential composition*, *juxtaposition* and *feedback* and four connectors – *identity, transposition, identification* and *ramification*.

The node operators and the connectors have a rich set of algebraic properties that basically reflect our visual intuition about flow graphs. In the following we review these properties and point out how they fit in the general setting of category theory. However,

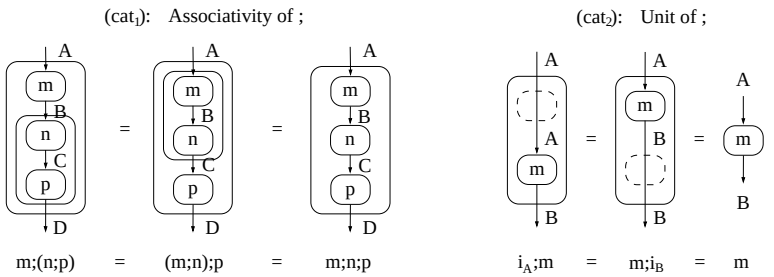

Figure 6.3 Properties of sequential composition

as Figures 6.3, 6.4 and 6.5 clearly show, no background in category theory is really necessary to understand them and the interested reader may consult [GBSS98] for a complete treatment. Moreover, these properties are not a prerequisite to understand the rest of the paper. We give them here basically for reference purpose and to establish the connection with other work in the semantics of concurrent processes. They are also needed in the proofs of similar properties for interaction graphs.

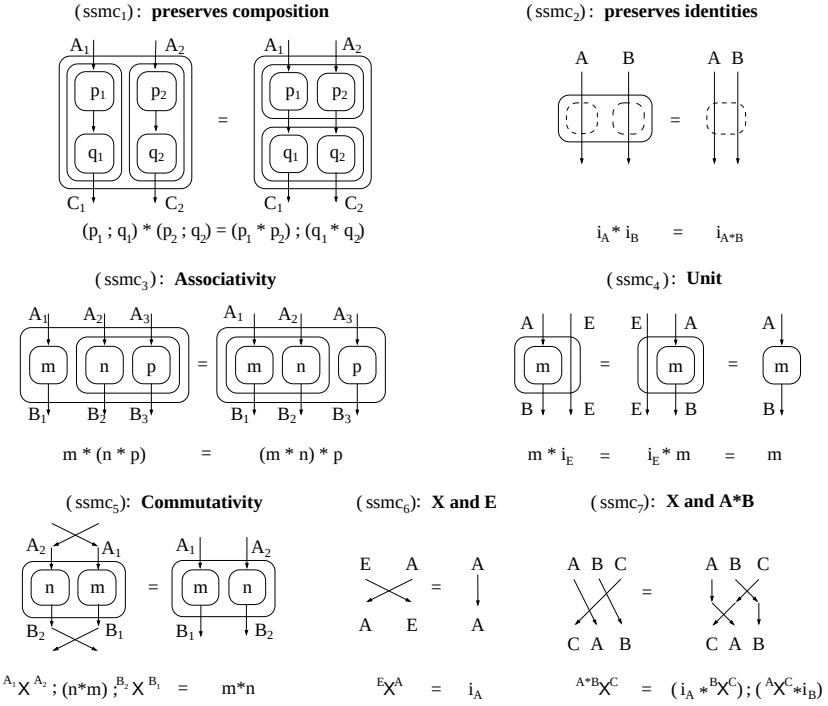

Figure 6.4 Properties of visual attachment

As shown in Figure 6.3, sequential composition is associative and has identities as neutral elements. Hence, in mathematical terminology, nodes equipped with sequential composition define a *category*. As shown in Figure 6.4, juxtaposition is defined both on arrows and on nodes in such a way that it preserves identities and composition. In mathematical terminology, it is a *functor*. Moreover, it is associative, has a neutral

element and commutes with transposition. It therefore defines a strict *symmetric monoidal category*. As shown in Figure 6.5, feedback also allows to construct loops and its properties extend the strict symmetric monoidal category to a *trace monoidal category*.

The *tightening* equation allows to tighten the scope of the feedback. In mathematical terminology one says that the feedback $\uparrow^C_{A,B}$ is *natural* in the arrows A and B, i.e., in the arrows that are not fed back. The *sliding* equation allows to slide n along the feedback loop. In mathematical terminology the feedback operator $\uparrow^C_{A,B}$ is *natural* also in the feedback arrow C. The *superposing* equation says that feedback superposes over visual attachment and the *yanking* equation shows how feedback relates to transposition. Finally, the *vanishing* equations show how to decompose the feedback loop.

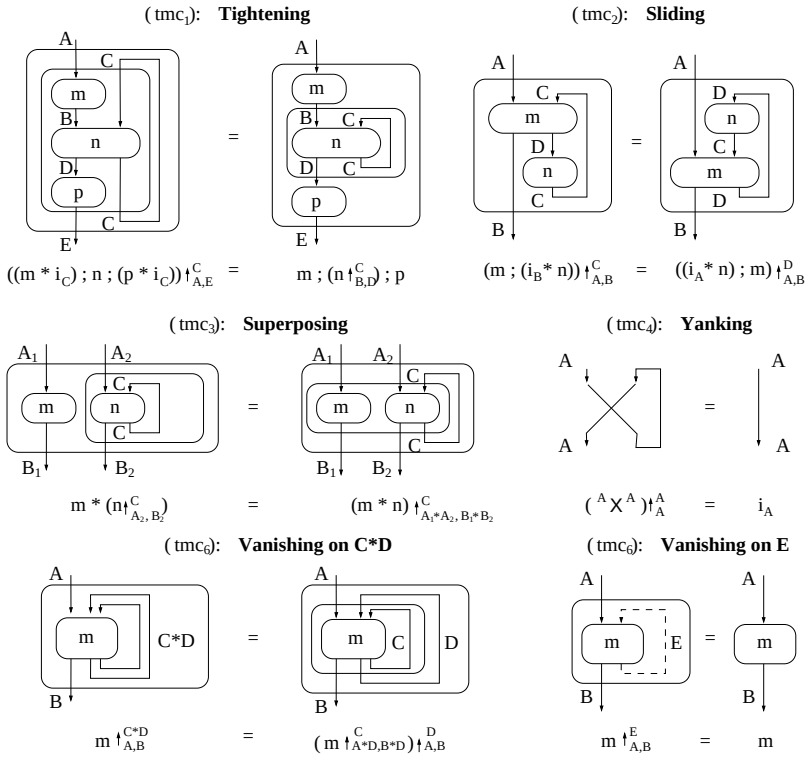

Figure 6.5 Properties of feedback

Identification is associative, has neutral elements and commutes with transposition. Hence, it mathematical terms, it defines a *monoid structure* on each arrow. Ramification is coassociative has neutrals element and commutes with transposition. Mathematically speaking, it defines a *comonoid structure* on each arrow. Moreover, identification preserves ramification and the other way around, i.e., identification is a comonoid morphism and ramification is a monoid morphism. Hence they define a *bimonoid structure* on each arrow.

3. INTERACTION GRAPHS

Flow graphs are a very basic formalism that allows us to describe the structure of any interactive system. However, they are too low level to directly cope with the constructs occuring in UML-RT. In particular they cannot directly express UML-RT *protocol types*, *duplex ports* and *duplex channels*.

For example, consider the following UML-RT protocol type `Tel` between a telephone and its associated telephone driver, *defined from the point of view of the driver*:

protocol `Tel` = { **input** = {`offH, onH, tlk, (dig, N)`}
output = {`dtB, dtE, rtB, rtE, tlk`}}

where `offH, onH, tlk, dig, dtB, dtE, rtB` and `rtE` stand for off hook, on hook, talk, digit, dial tone begin/end and ring tone begin/end respectively. The digit message contains additional data $x \in N$ where $N = \{0, \ldots, 9\}$. Then the interconnection between the telephone and the driver is given in a UML-RT capsule diagram as shown in Figure 6.6. The driver contains a duplex port `d` of type `Tel` and

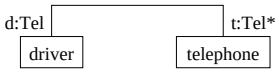

Figure 6.6 A simple telephone architecture in UML-RT

the telephone contains a duplex port `t` having the *dual type* `Tel*`. By duality it is meant that input and output are interchanged. These ports are connected by a duplex bend channel whose ends have dual type. By convention, duplex (or bidirectional) channels are drawn without any arrow head.

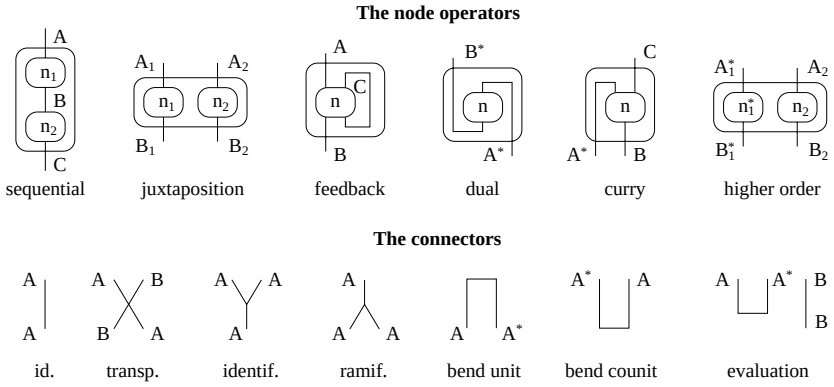

Figure 6.7 The node operators and the connectors

To model this structure directly, we need a graph formalism that includes both duality and bend connectors with dual ends. We call such a graph formalism *interaction graphs.* The reason for this name is that each arrow in these graphs defines a *two way* communication. In other words, it defines an *interaction*. As shown in Figure 6.7, the mere introduction of duality and of bend connectors with dual ends immediately allows to define six operators on nodes – *sequential composition*, *juxtaposition*, *feedback*,

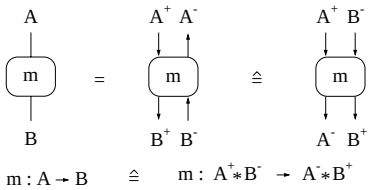

Figure 6.8 Translation of interaction graphs nodes to flow graph nodes

dual, *curry* and the *higher order nodes* constructor and seven connectors – *identity*, *transposition*, *identification*, *ramification*, *bend unit*, *bend counit* and *evaluation*.

As we show in the next sections, interaction graphs are implemented by flow graphs in the same way higher order programming languages are implemented by assembler languages. Hence, each interaction graph could be analyzed by translating it to a flow graph. However, this analysis could be very cumbersome. It is therefore much better to prove once and for all a set of core properties for interaction graphs which allows to manipulate them directly in a similar way to flow graphs. In fact, *all properties* of flow graphs naturally extend to interaction graphs. Moreover, the introduction of bend connectors and dual types is the source of new graph operators and connectors and of a rich set of new properties. They allow us to adjust the component interaction interfaces and to dynamically manipulate both procedures and capsules.

In the following sections we show the implementation of interaction graphs in terms of flow graphs and the new properties of interaction graphs. Both the proofs and the properties similar to flow graphs are not given because of obvious space limitations.

3.1 ARROWS

Let X and Y be flow graph arrows. An *interaction graphs arrow* is defined as a *pair* (X, Y) of flow graph arrows where X has a top down orientation and Y has a bottom up orientation. By using such pairs, interaction graphs can deal simultaneously with three kinds of arrows: *unidirectional top down* arrows, *unidirectional bottom up* arrows and *bidirectional arrows*. Unidirectional top down arrows are pairs (X, E) where E is the flow graphs empty arrow. Unidirectional bottom up arrows are pairs (E, X) and bidirectional arrows are pairs (X, Y). To simplify notation and to make the connection between the components of the pair more explicit we denote interaction graphs arrows by A, B etc., and annotate their top down and bottom up components with a *plus* and respectively a *minus* sign. Hence we write A for (A^+, A^-). Visually, we distinguish bidirectional arrows from unidirectional arrows, by drawing them without any head. Moreover, since everything proved for bidirectional arrows also holds for unidirectional arrows we shall work in the following sections only with bidirectional arrows.

3.2 NODES

A node $n : A \to B$ in an interaction graph maps the interaction arrow $A = (A^+, A^-)$ to the interaction arrow $B = (B^+, B^-)$, as shown in Figure 6.8, middle. The interaction arrow A is the *client interface*. The interaction arrow B is the *server interface*. Going from flow graphs to interaction graphs is actually a switch of programming paradigm. We abandon the *input/output* paradigm in favor of the *client/server* paradigm.

However, since as in programming, each interaction graphs arrow is defined by a pair of flow graphs arrows, for each interaction graphs node there is a corresponding flow graphs node as shown in Figure 6.8, right. We take these flow graphs nodes as the *implementation* (or representation) of the interaction graphs nodes. When looking at the implementation of an interaction graphs connector in the next sections, keep in mind this translation because it determines a unique interface of the corresponding flow graphs connector.

Using this implementation, each interaction graph may be implemented by a flow graph. Moreover, as we see later, each interaction graphs operator may be implemented by a composition of flow graphs operators. Formally, implementing an interaction graph node $m : (A^+, A^-) \to (B^+, B^-)$ by a flow graphs node $m : A^+ * B^- \to A^- * B^+$ can be described by a representation function (or relation) rep such that:

$$rep(m : (A^+, A^-) \to (B^+, B^-)) \;=\; m : A^+ * B^- \to A^- * B^+$$

Abstracting (or embedding) a flow graphs node $m : A^+ * B^- \to A^- * B^+$ to an interaction graphs node $m : (A^+, A^-) \to (B^+, B^-)$ can be formally described by an *abstraction* function (or relation) abs such that:

$$abs(m : A^+ * B^- \to A^- * B^+) \;=\; m : (A^+, A^-) \to (B^+, B^-)$$

Taking $abs(A) = (A, E)$ each flow graph may be embedded into an interaction graph with arrows pointing only top down (with the exception of the feedback arrow). As a consequence, each flow graph node operator and each flow graph connector has a corresponding interaction graph node operator and connector, respectively. Instead of inventing for each a new notation, we use the same symbols in the interaction graphs, too. In other words, we *overload* the node operators and connectors. Moreover, instead of dealing explicitly with the representation function, i.e., instead of writing $rep(m : (A^+, A^-) \to (B^+, B^-)) = m : A^+ * B^- \to A^- * B^+$ we shall simply write $m : (A^+, A^-) \to (B^+, B^-)) \mathrel{\hat{=}} m : A^+ * B^- \to A^- * B^+$, which can be read as a *definition*. Since in most cases we will not make the types explicit one should keep in mind that the symbol on the left of the definition is an interaction graphs symbol, whereas the same symbol on the right of the definition is a flow graphs symbol.

3.3 OPERATORS ON NODES

Sequential Composition. As shown in Figure 6.9, left, the most basic way to *connect* two interaction graphs nodes is *sequential composition*. By this we connect the server interface of one node to the client interface of the other node, if they have the same type. Textually we denote this operator also by ; (hence ; is an overloaded operator). As shown in Figure 6.9, right, given $m{:}A{\to}B$ and $n{:}B{\to}C$ we define (implement) the interaction graphs composition $m{;}n{:}A{\to}C$ in terms of the flow graphs sequential composition, transposition and feedback. As in flow graphs, the composition in interaction graphs defines both a *connection* and a *containment* relation.

Using the properties of flow graphs one can easily show that sequential composition is associative and has the identities as neutral elements. Hence interaction graphs nodes equipped with sequential composition define a *category*.

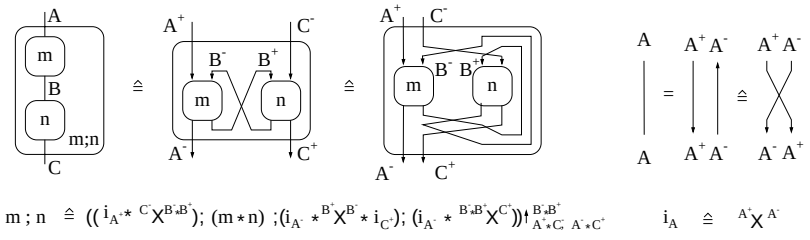

Figure 6.9 Sequential composition and identity

Juxtaposition. By *juxtaposition* we mean that nodes and corresponding arrows are put one next to another, as shown in Figure 6.10, left. To obtain a textual representation

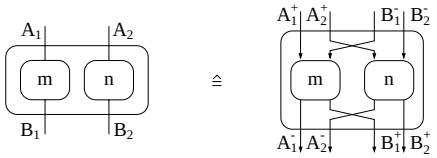

Figure 6.10 Juxtaposition

for juxtaposition, we need therefore a juxtaposition operator defined both an arrows and on nodes. We denote this operator also by $*$ (hence $*$ is an overloaded operator). Given two arrows A_1 and A_2 their juxtaposition is expressed by $A_1 * A_2$ and defined in terms of flow graphs juxtaposition by $(A_1^+ * A_2^+, A_1^- * A_2^-)$. Given two nodes $m{:}A_1 \to B_1$ and $n{:}A_2 \to B_2$ their juxtaposition is expressed as $m*n{:}A_1 * A_2 \to B_1 * B_2$ and defined, as shown in Figure 6.10, in terms of flow graphs juxtaposition. Similarly to sequential composition, juxtaposition also defines a containment relation.

As with flow graphs, 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 modulo a *transposition* isomorphism ${}^{A}\mathsf{X}^{B} : A*B \to B*A$. We define the interaction graphs transposition ${}^{A}\mathsf{X}^{B}$ in terms of flow graphs transposition as shown in Figure 6.11, right.

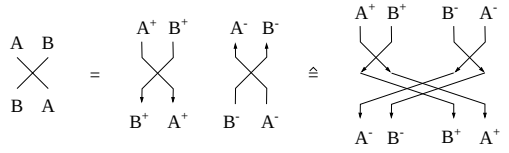

Figure 6.11 Interaction graphs transposition

Using the properties of flow graphs it is easy to show that juxtaposition is defined both on arrows and on nodes such that it preserves identities and composition. Hence it

is a *functor*. Moreover, it is associative, has a neutral element and commutes with transposition. It therefore defines a *strict symmetric monoidal category.*

Duality and bend connectors. Each line A in the interaction graphs is defined by a pair (A^+, A^-) of flow graphs arrows, with A^+ pointing in top down direction and A^- pointing in bottom up direction. It makes therefore sense to think of reversing the direction of these arrows. This is accomplished by a *duality operator* $(.)^*$ defined as follows: $A^* = (A^+, A^-)^* = (A^-, A^+)$. Duality is involutive since $A^{**} = A$.

In order to extend duality to nodes we need two bend connectors: the *bend unit* $\sqcap_A : \mathbb{E} \rightarrow A * A^*$ and the *bend counit* $\sqcup_A : A^* * A \rightarrow \mathbb{E}$. The bend unit and counit are

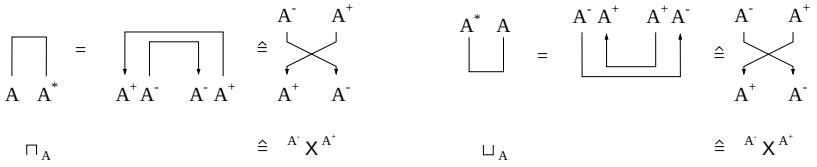

Figure 6.12 Unit and counit bends

defined, as shown in Figure 6.12, by using the flow graphs transposition connector. It is interesting to note that $\sqcap_A$ and $\sqcup_A$ have the same definition (or implementation) in the flow graphs category. However, their type in the interaction graphs category is different which means that they are used in different contexts.

Nodes duality. Using the bend connectors one can extend the duality operation to nodes. Given $m : A \rightarrow B$ we define $m^* : B^* \rightarrow A^*$, as shown in Figure 6.13, left. The

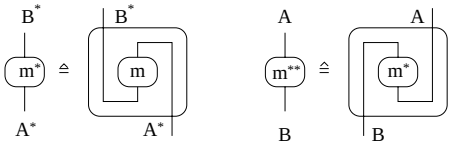

Figure 6.13 Duality extended to nodes

duality on nodes is also involutive if $m^{**} = m$. As shown in Figure 6.13, this is indeed the case if the unit $(igCcc_1)$ and counit $(igCcc_2)$ axioms in Figure 6.14 hold. This can be easily checked by using their definition. Moreover $(igCcc_1)$ and $(igCcc_2)$ also say that $(i_{A^*})^* = i_A$ and $(i_A)^* = i_{A^*}$, i.e., that duality preserves identities. It is also easy to check that the involutive duality operators preserves sequential composition. Hence, in mathematical terminology, duality is an *involutive functor*. Since this functor reverses the direction of arrows and consequently the direction of composition, it is called a *contravariant* functor. The contravariant duality functor enriches the interaction graphs with the structure of a *compact closed category* (abbreviated as igCcc). This structure is very rich and allows us to define feedback, curry and the data nodes constructor as derived operators.

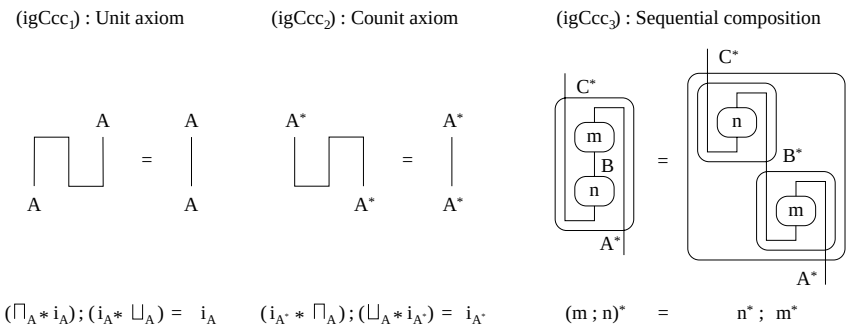

Figure 6.14 Functor properties of duality

Nodes as Data. Using the bend connectors one can move any client arrow of an interaction graphs node to its server side and any server arrow to its client side. In particular, one can move all arrows either on the client side or on the server side. Moving all arrows on the client side one obtains nodes very similar to the architecture nodes in UML-RT. In this case, it is possible to connect two nodes only by using the bend unit and the transposition connectors. As a consequence, the connected arrows are requested to be *dual*. This is exactly what the graphical editor for UML-RT architecture diagrams checks.

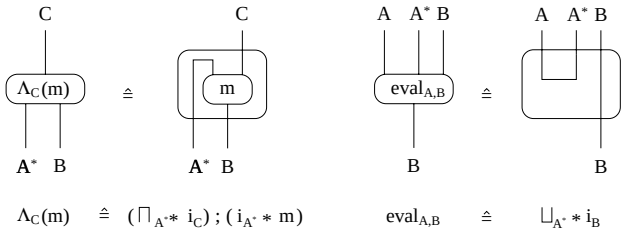

Figure 6.15 Curry isomorphism and eval connector

Moving all the arrows of an interaction graphs node to its server side transforms this node into a *data node*. Why is it a data node? Because its client interface becomes empty and therefore no other client node may be sequentially connected to it anymore. To use this node one needs therefore an evaluation operator, which applies the data node to its input (or routes the data to the input). However, since changing the interface of a node is merely a matter of convenience, the visual intuition tells us that between real nodes and data nodes there should be a *one-to-one* correspondence.

The correspondence is given by an isomorphism $\Lambda_C{:}(A{*}C{\to}B){\to}(C{\to}(A^{*}{*}B))$, known as *curry* and a distinguished connector $eval_{A,B} : A{*}(A^{*}{*}B) \to B$, as shown in Figure 6.15. Their definition is as expected a straight forward use of the bend connectors. Using the properties of these connectors, one can easily check that the relation between $curry$ and $eval$ is the one shown in Figure 6.16. The β-axiom says that *bending* an input arrow of a node twice does not change the meaning of the node. The η-axiom says that *bending* an output arrow of a node twice does not change the meaning of a node. The λ-calculus equivalent for the β-axiom is

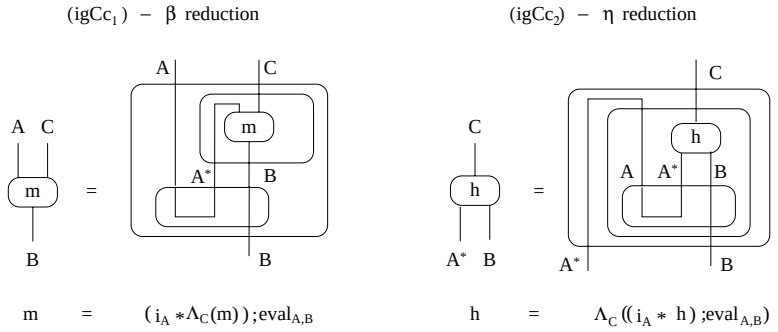

Figure 6.16 The curry-eval axioms

$m(a,c) = eval(a, \lambda x.m(x,c))$. Since $eval(a,f)$ is often written as $f(a)$ the above equation is also written as $m(a,c) = (\lambda x.m(x,c))(a)$. The λ-calculus equivalent of the η-axiom is $h = \lambda x.eval(x,h)$ or with the above convention $h = \lambda x.h(x)$. The identifier x is supposed to not occur free in h.

Higher order nodes. An element of $A^* * B$ is a data node $f : \mathbb{E} \to A^* * B$. In order to emphasize that $A^* * B$ contains data nodes we also use for it the arrow like notation $A \multimap B$. Hence $\multimap$ maps a pair of interaction arrows A and B to an interaction arrow $A \multimap B$. But what about nodes?

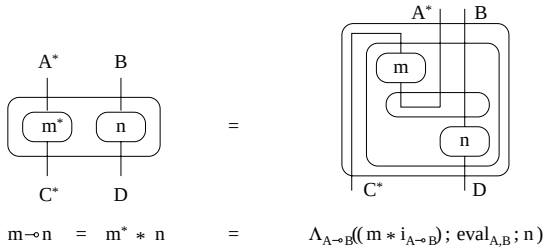

Figure 6.17 The definition of $\multimap$

Consider a pair of nodes with arrows in opposite (dual) directions, as shown in Figure 6.17, left. Textually, we denote this construction by $m \multimap n : (A \multimap B) \to (C \multimap D)$. It takes *data nodes* in $A \multimap B$ as input and delivers data nodes in $C \multimap D$ as output. Using $curry$ and $eval$ the meaning of $m \multimap n$ is defined as shown in Figure 6.17, right. Since $A \multimap B$ is defined in terms of juxtaposition and duality, it inherits the properties of both these operators, as shown in Figure 6.18. In mathematical terminology, $\multimap$ is a *functor* changing the direction of arrows in the first argument (the argument where duality is applied). This functor is a consequence of the compact closure of interaction graphs and the structure it determines together with *eval* and *curry* on these graphs is that of a *closed category* (igCc stands for interaction graphs closed category). Closed categories are models for the *λ-calculus*, the calculus underlying all higher order functional programming languages.

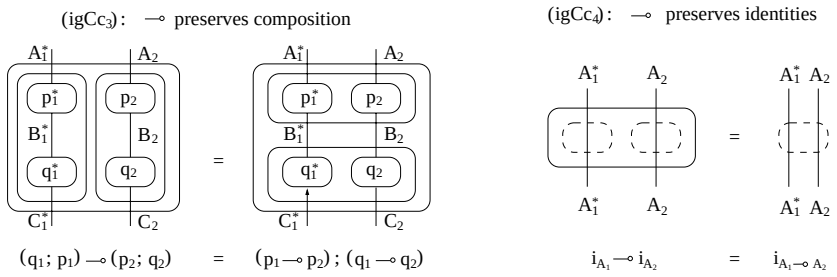

Figure 6.18 The functor axioms for ⊸

Feedback. The bend connectors allow us to define the *feedback operator* for interaction graphs in a very simple way as shown in Figure 6.19. Using the properties of

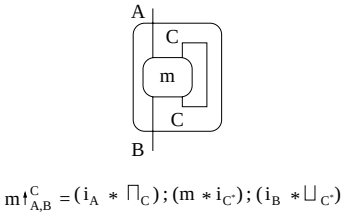

Figure 6.19 Feedback in a compact closed category

the bend connectors it is easy to show that the interaction graphs feedback also defines a *trace monoidal category* structure on the underlying category (see [GBSS98]).

3.4 CONNECTORS

In the previous sections we already introduced four connectors: *identity* i_A, *transposition* ${}^A\mathsf{X}^B$, *bend unit* $\sqcap_A$ and *bend counit* $\sqcup_A$. While the bend connectors are new, the identity and transposition connectors are extensions of the analogous flow graphs connectors to interaction graphs. A similar extension is possible for the *identification* connector $\vee_A$ and the *ramification* connector $\wedge^A$.

Identification. The definition of the interaction graphs *identification* connectors $\top_A : E \to A$ and $\vee_A : A * A \to A$ is shown in Figure 6.20. It is based on flow graphs identification, ramification and transposition. Using the properties of flow graphs it is

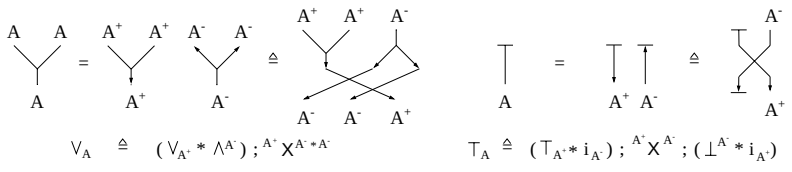

Figure 6.20 Interaction graphs identification

easy to show that identification is associative, has neutral element and commutes with transposition. Hence, it defines a *monoid structure* on each arrow.

Ramification. The definition of the interaction graphs *ramification* connectors $\perp^A : A \to E$ and $\wedge^A : A \to A * A$ is shown in Figure 6.21. It is based on flow graphs identification, ramification and transposition. Using the properties of flow graphs it

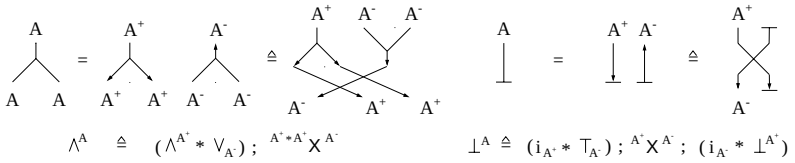

Figure 6.21 Interaction graphs ramification

is easy to show that ramification is coassociative has neutral element and commutes with transposition. Hence, it defines a *comonoid structure* on each arrow. Moreover, identification preserves ramification and the other way around, i.e., identification is a comonoid morphism and ramification is a monoid morphism. Hence they define a *bimonoid structure* on each arrow.

4. CONCLUSIONS

The main benefits of the graph theory defined in this paper 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 interface to tool developers.

The calculus of flow graphs (see [GBSS98]) is a simpler, more general and more intuitive presentation of the one given in the context of flow charts in [CS90]. Models of flow graphs were studied independently in the context of flow charts in [CS90] and in the context of data flow networks in [B87, GS96]. In [GSB98a] we show how to combine these models to obtain a semantics for ROOM. This semantics is extended for the hierarchical specification of hybrid systems in [GSB98b].

The calculus of interaction graphs is a simpler, more intuitive and more general formulation of interaction categories (see [AGN94]) that uses ideas from [JSV96]. It is not only closer to UML-RT but, in our opinion, a better foundation for the theory of typed concurrent systems. This was only possible by having in mind the concrete implementation of UML-RT. A semantic model for this calculus is given in [GBSR99]. It is also inspired by the UML-RT implementation. Since it defines a game semantics for interaction graphs, it is very general too and it may be used to understand concurrency.

The use of the interaction graphs theory in the context of visual formalisms, in particular for UML-RT is new. It clearly shows that visual formalisms are not only intuitive but also can have a deep underlying formal theory. In fact, we are using this theory to compare, optimize and execute designs. Moreover, it is the best starting point for verification techniques. In fact, we use this theory in the project Mocha

(see [AHM+98]) for modular model checking. More generally, because of the deep connection between interaction graphs and linear logic (see [A96]) other analysis and verification techniques may be used as well.

References

[A96] S. Abramsky. Retracing some paths in process algebra. In *Seventh International Conference on Concurrency Theory (Concur'96), Lecture Notes Computer Science 1055*, pages 21–33, 1996.

[AGN94] S. Abramsky, S. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. To appear in Proc. Marktoberdorf Summer School, 1994.

[AHM+98] R. Alur, T. A. Henzinger, F.Y.C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. To appear in the Proceedings of the Tenth International Conference on Computer-aided Verification (CAV 1998), Lecture Notes in Computer Science, Springer-Verlag, 1998.

[B87] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[CS90] V.E. Căzănescu and Gh. Stefănescu. Towards a new algebraic foundation of flowchart scheme theory. *Fundamenta Informaticae*, 13:171–210, 1990.

[GBSR99] R. Grosu, M. Broy, B. Selic, and B. Rumpe. A formal foundation for UML-RT. To Appear, July 1999.

[GBSS98] R. Grosu, M. Broy, B. Selic, and Gh. Stefanescu. Towards a calculus for UML-RT specifications. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications, Vancouver, Canada, Monday, October, 19th*. TUM-I9820, 1998.

[GS96] 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 Methodology and Software Technology, AMAST'96, Munich*, pages 505–519. LNCS 1101, 1996.

[GSB98a] R. Grosu, Gh. Stefanescu, and M. Broy. Visual formalisms revisited. In *CSD '98, International Conference on Application of Concurrency to System Design, Aizu-Wakamatsu City, Fukushima*. IEEE, March 1998.

[GSB98b] Radu Grosu, Thomas Stauner, and Manfred Broy. A modular visual model for hybrid systems. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*. Springer-Verlag, 1998.

[JSV96] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.*, 119:447–468, 1996.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Available under http://www.objectime.com/uml, April 1998.

About the Authors

Radu Grosu studied computer science at the Technical University of Cluj. In 1994 he received his PhD at the Technical University of Munich where he became a scientific assistant. Since 1998 he is also a visiting researcher at the University of Pennsylvania. His research interests include theoretical and practical aspects in the design and analysis of reactive, real-time and hybrid systems. In particular he is interested in the formal foundation of the software engineering's visual formalisms and worked in several projects on this issue. Currently at the University of Pennsylvania he is investigating the analysis potential of visual formalisms both for reactive and for hybrid systems.

Manfred Broy is full professor of computing science at the Technical University of Munich. His research interests are theory and practice of software and systems engineering aspects includind system models, specification and refinement of system components, specification techniques, development methods and verification. He leads a research group working in a number of industrial projects that try to apply mathematically based techniques and to combine practical approaches to software engineering with mathematical rigor. Professor Broy is a member of the European Academy of Sciences. In 1994 he received the Leibniz Award by the Deutsche Forschungsgemeinschaft.

Bran Selic is Vice President of Advanced Technology, at ObjecTime Limited. He has over 25 years of experience in real-time software, focussing on distributed systems and object-oriented development. He is the principal author of the textbook, "Real-Time Object-Oriented Modeling" and is a core member of the team that defined the Unified Modeling Language (UML), a standard for object-oriented analysis and design that was issued by the Object Management Group (OMG). Bran is currently working on applying UML to real-time development. He is also co-chair of the Real-Time Analysis and Design Working Group within the OMG.

Gheorghe Stefănescu completed his university studies at the University of Bucharest, receiving a PhD degree in 1991. He spent 15 years as a researcher at the Institute of Mathematics at the Romanian Academy. Currently he is Professor of Computer Science at the University of Bucharest. He was visiting professor/researcher at various universities, including Technical University Munich, Kyushu University, University of Amsterdam, and Utrecht University. His main research interests are in formal methods applied to distributed computing and object-oriented systems, especially using algebraic methods.