Aspects + GAMMA = AspectGAMMA
A Formal Framework for Aspect-Oriented Specification

Mohammad Mousavi [a] Giovanni Russello [a] Michel Chaudron [a]
Michel Reniers [a] Twan Basten [a] Angelo Corsaro [b]
Sandeep Shukla [b] Rajesh Gupta [b] Douglas C. Schmidt [b]

[a] *Technische Universiteit Eindhoven (TU/e)*
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
Email{m.r.mousavi,g.russello,m.r.v.chaudron, m.a.reniers,a.a.basten}@tue.nl

[b] *University of California at Irvine*
Irvine, CA 92697
Email{corsaro@ece, skshukla@ics,rgupta@ics, schmidt@ece}.uci.edu

---

Center for Embedded Computer Systems at UCI Technical Report # CECS-02-01

Contents

1	Introduction	1
2	Exploring Aspects	2
2.1	Modeling Computation with GAMMA	3
2.2	Coordination	6
2.3	Timing	7
2.4	Distribution	8
3	Weaving Aspects	9
4	Related Work	11
5	Conclusion and Future Research	12

List of Figures

1	Basic GAMMA Syntax	4
2	GAMMA Program for the Elevator System	6
3	Fragment of the Timed Transition System after Weaving.	11

List of Tables

Abstract

This paper describes an extension to the GAMMA formalism, which we name AspectGAMMA, and we show how non-computational aspects can be expressed separately from the computation in this framework. Examples of such aspects include real-time constraints, location/distribution aspects, behavioral requirement aspects, fault-tolerance aspects, power requirement aspects, and many others. The idea is to abstract the emerging idea of aspect-oriented programming (AOP) into a formal framework, thereby facilitating specification-driven design, enabling formal validation, and design reuse at the requirement specification level. The main goal of this position paper is to outline the way towards a formal foundation of aspect-oriented specification and refinement towards implementation.

1 Introduction

Separation of concerns is one of the concepts at the core of modern software design and evolution. It has been advocated as a key principle for reducing the complexity of developing large-scale software systems [13]. Different techniques and methods have been proposed that help to separate concerns, but some types of concerns, such as timing and distribution, remain hard to separate. These concerns usually *crosscut* the responsibility of several encapsulation units, such as classes in the context of object-oriented programming languages like Java or C++.

One of the fundamental tenets of aspect-oriented programming (AOP)[11] is that complexity in the design of computer programs arises from the fact that many individual concerns (aspects) from the (user) requirements domain are ultimately scattered across multiple locations in the solution (the program). At the same time, however, these parts must be mutually consistent/compatible, and evolve consistently during further changes [6]. Experience has shown that separating these different concerns explicitly in the software design helps developers manage software complexity more effectively than tangling the concerns into tightly coupled programs. AOP languages, such as AspectJ [15], provide linguistic support for advanced separation of concerns by modeling criteria that correspond to different requirement/design concerns [6].

However, linguistic mechanisms in an implementation language do not help in using the aspect orientation in requirement capturing and other phases of the software engineering lifecycle that precede implementation. As a result, there must be a formal modeling framework that allows requirements to be specified

in a formal way that enables the separation of different aspects. To enhance reusablity of components, aspects specifications should be independent of each other. The computation itself might also be decomposed into different aspects. This is a basic principle in aspect orientation, the so-called *obliviousness property* [6]. This property suggests that an aspect modeling language should not be aware of other (independent) aspects present in a software architecture. As a result, it should not address the issues of other aspects, to the greatest possible extent. Starting from this assumption, we can deduce that an ideal aspect model would be a simple and tailor-made model that only focuses on a particular well-defined aspect domain. The current requirement specification language/formalism paradigms, such as Object Z [8], however, largely contain a complex and entangled mixture of different aspects.

In this paper, we discuss the main characteristics of an aspect-oriented formal specification framework, which is based on a multiset transformation language called GAMMA, which is a formalism based on multiset rewriting [3, 2]. We illustrate how having a tailor-made formalism for each aspect that is abstracted from other aspects is a key benefit of such a formal design framework. To clarify our discussions, we sketch an architecture specification and design method for reactive distributed real-time embedded systems (DRE systems).

In the approach we describe in this paper, we propose separating the concerns of computation, coordination, timing, and distribution, through different simple and abstract notations for these aspects. We also describe a weaving process that maps all these different aspects to a single semantic domain. The method is based on a formal semantics that should ultimately enable automated reasoning about designs. The idea exploited in this method can be extended to other aspects, and extended with more complex weaving criteria.

The remainder of this paper is organized as follows: Section 2 discusses computation, coordination, timing, and distribution as different aspects of a software design and suggest languages/notations to specify them. Section 3 contains a simple model of weaving the functional and non-functional aspects in a single semantics model. Section 4 proceeds with previous and related work, and Section 5 provides concluding remarks and research directions.

2 Exploring Aspects

This section focuses on the specification of computation and the three aspects coordination, timing, and distribution. We use a subset of GAMMA for specifying basic component functionalities (computations) and present its distinguishing features. We then present some ideas about specifying other

aspects.

2.1 Modeling Computation with GAMMA

GAMMA is an abstract language, based on multiset rewriting on a shared data-space, designed to support parallel execution of a program on parallel and/or distributed architectures [3, 2]. The basic and atomic piece of functionality in GAMMA is the *rule*. The calculus of GAMMA [9] contains some composition operators to compose rules. In [3], some patterns of rule composition (called *tropes*) are suggested to give hints for a program designer on how to compose/decompose functionalities to construct specific programs.

In this section, we focus on a basic subset of GAMMA involving the specification of basic rules. We then factor out structuring decisions and make them a separate aspect model, the coordination model. The high level of abstraction proposed here follows from our basic assumption that a functionality specification model should only address essential computation for achieving the required computational functionalities. The other requirements which are non-functional, such as distribution, timing, etc., should be kept independent from functional requirements, as well as others so that change and evolution of it is localized and does not influence other parts of this or other aspects. This separation of basic functionality from coordination can also enhance reusability since a single functionality model may be reused with different coordination models to construct different programs or versions of a single program with different levels of efficiency [4]. For example, if you are expressing the requirements for an elevator control, the basic computation aspect includes the various signals and how they should behave with respect to a clock. The distribution will localize some of the signals into different locations, and distribute the corresponding signal value computations. If signal values are shared across location, the distribution aspect might express protocols for such shared accesses. The timing aspect, will describe constraints on signal behaviors.

Henceforth, the GAMMA model is only considered with basic functionalities in the form of a simple input-computation-output pattern that abstracts from the following details:

(1) *Relative ordering of actions (coordination).* Basic functionalities (called rules) are specified independently of each other. Hence, no special ordering of actions (control structure) is imposed on this particular specification.
(2) *Timing.* The basic GAMMA model does not include any information about timing. Since it abstracts from ordering of actions, even a qualitative (causal) notion of time is not present in the GAMMA model.

(3) *Distribution.* For any distributed system, the shared data-space is an abstraction that eases the programming, yet must be distributed in the implementation.
(4) *Fault tolerance.* The GAMMA execution model requires programs to be designed in such a way that duplicated execution of atomic actions of a program cannot affect the functionality. Hence, replication of actions can be added transparently to the functional model.

The abstract nature of GAMMA in exploiting independent rewrite rules makes it suitable for definition of basic functionalities of software components. This does not mean that any of the above concerns are unimportant in system design and could be neglected completely. Conversely, this abstractness provides the desired orthogonality, so that any of the above items can be specified and maintained as a separate aspect.

The syntax of a simple GAMMA program is given in Figure 1.

$$
\begin{array}{lll}
Program & ::= & ProgramName = \{Rules\} \\
Rules & ::= & Rule \mid Rule, Rules \\
Rule & ::= & RuleName = MultisetExp \mapsto MultisetExp \Leftarrow Condition \\
MultisetExp & ::= & \epsilon \mid BasicExp \mid BasicExp, MultisetExp \\
BasicExp & ::= & Variable \mid (Variable, BasicExp)
\end{array}
$$

Fig. 1. Basic GAMMA Syntax

A GAMMA program consists of a non-empty set of rules, each rewriting the content of the shared multiset of data items. Execution of a program consists of applying rules to the multiset in all possible orders (sequential and parallel). Each rule consists of a set of terms valuated by multiset content values (this replacement is not necessarily unique for a specific rule and multiset). If a certain valuation of variables satisfies the condition in a rule, applying the rule results in removing the left-hand side valuations from the multiset and replacing them by the valuation of the right-hand side expression. We do not present the exact syntax of such logical formulas in this paper – allowing them to be defined by selecting an appropriate underlying logic. However, we use the syntax and semantics of predicate logic formulas throughout this paper.

In [12], a formal operational semantics for GAMMA is given in the style of

Plotkin [14]. Execution of a GAMMA program is based on execution of its individual rules. Hence, the main observable events in the semantics of such a program are changes in the shared multiset (multiset substitutions) and the termination of rules in case there are no enabling values to be found in the shared multiset.

In the given semantics, the tuple $\langle R, M\rangle$ represents the state of the system by the GAMMA rules and multiset contents. The deduction rule **(Term)** in GAMMA semantics shows the termination of a single rule. A rule terminates if and only if there is no substitution of its variables from the multiset content such that they satisfy the rule condition (denoted by $M, \overline{v} \propto r$).

The deduction rule **(Trans)** represents a single rule execution, which is a substitution of the valuation of left-hand side expression by the right-hand hand side one. In **(Par)** (for parallel execution of rules), notion of independence ($\bowtie$) is introduced. Two rules can execute in parallel if and only if they have disjoint valuations for their write sets. Independence condition holds for two rule if and only if their write sets are disjoint. To be more precise:
if $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$ then

$$M \models \sigma_1 \bowtie \sigma_2 \stackrel{\triangle}{=} N_2 \sqsubseteq M[\sigma_1] \ \wedge \ N_1 \sqsubseteq M[\sigma_2]$$

The formal semantics defines the rules for executing single rules. It does not, however, suggest any order of rules to execute a program and therefore abstracts away from the behavioral aspect of design. This suggests that execution of a naive GAMMA program bears any chaotic order on its rule executions.

Example 1 *An elevator system, functionality specification.* Our elevator system consists of an elevator moving up and down between floors of a building (numbered from 0 to $MaxFloor$) to service requests. On each floor there is a push button to announce a request for an elevator when turned *on*. When an elevator arrives on a floor, the request flag is turned *off* automatically. The same setting works for the push buttons inside the elevator, which indicate the requested stops for passengers inside.

To model this distributed real-time system we propose a multiset containing events requesting an elevator stop represented by $((inStop, i), status)$ and $((extStop, i), status)$ that show the status of the request button for the i'th floor, inside and outside the elevator, respectively. The tuple (cf, i), shows where the elevator currently resides. The GAMMA program for the elevator system is given in Figure 2.

$$
\begin{aligned}
ElevatorSystem = \{\ & inRequest = ((inStop, i), off) \mapsto ((inStop, i), on), \\
& extRequest = ((extStop, i), off) \mapsto ((extStop, i), on), \\
& moveUp = (cf, i) \mapsto (cf, i+1) \Leftarrow \exists j; i < j \wedge ((extStop, i), off) \in State \wedge ((inStop, i), off) \in State, \\
& moveDown = (cf, i) \mapsto (cf, i-1) \Leftarrow \exists j; j < i \wedge ((extStop, i), off) \in State \wedge ((inStop, i), off) \in State, \\
& load = ((extStop, i), on) \mapsto ((extStop, i), off) \Leftarrow (cf, i) \in State, \\
& unload = ((inStop, i), on) \mapsto ((inStop, i), off) \Leftarrow (cf, i) \in State \\
\}
\end{aligned}
$$

Fig. 2. GAMMA Program for the Elevator System

The initial multiset for this system is defined as:

$$
\begin{aligned}
State = [\ & ((inStop, 0), off), \ldots, ((inStop, MaxFloor), off), \\
& ((extStop, 0), off), \ldots, ((extStop, MaxFloor), off), \\
& (cf, 0) \\
],
\end{aligned}
$$

which shows that the elevator is at the ground floor initially and that there are no request for the elevator.

2.2 Coordination

The functionalities described by GAMMA programs allow for many meaningless executions of the system. In the elevator system, for example, the elevator can repeatedly move up and down between two floors without servicing any of the outstanding requests.

Regarding the model of separation of concerns proposed in this paper, we note that behaviour (both basic computations and coordination) is itself a complicated set of aspects that has been the main topic of discussion in the aspect-oriented community. As before, the abstractness of GAMMA is an elegant feature that allows us to define different composition, restriction, and ordering operators on basic rules.

To present our ideas about non-functional aspects, however, we present a simple and abstract model of basic functionalities in the GAMMA formalism, and do not discuss coordination in detail. The coordination of GAMMA programs is treated in [5] and for coordination of GAMMA programs including the timing aspect we refer to [12].

2.3 Timing

Timing constraints can be added to a specification to provide assertions regarding the execution time of GAMMA rules. This time is relative to the point from which the rule is selected for execution (when the previous rule execution is finished). We propose to add the timing aspect to a GAMMA specification by associating an interval to each rule name. This timing representation keeps the syntactic specification of timing separate from rule definitions, and hence allows independent change of both aspects. This method also allows a rule to have no timing assertion, which will be replaced by a default interval ($[0, \infty]$) in the weaving process.

Since GAMMA rules assume a shared access to data, the timing aspect does not specify any assumptions about the cost of accessing the data items in a distributed setting. The above estimation is therefore only related to the computation time for each functionality. In the next section, we investigate the effects of putting constraints on the sharing/distribution policy.

Example 2 *The elevator system, timing.* Suppose that the following timing information is given about the elevator system in Example 1:

- Pushing an internal or external button does not take time at all:

$$T_{inRequest} = T_{ExtRequest} = [0, 0].$$

- Going up and down between floors takes $StepTime$ for each floor:

$$T_{moveUp} = T_{moveDown} = [StepTime, StepTime].$$

- The elevator will be loaded/unloaded within $MinService$ and $MaxService$ amount of time, depending on the number of people and goods waiting for it:

$$T_{load} = T_{unload} = [MinService, MaxService].$$

The timing information allows us to verify the timeliness of a functional specification, possibly for a given coordination, assuming the aspects are appropriately weaved together. In Section 3, we explain how this timing information can be weaved together with the functional specification into a single semantic framework.

2.4 Distribution

As expressed in Section 2.1, GAMMA abstracts from distribution of data and processing and assumes a shared multiset. Moreover, the timing aspect does not refer directly to the distribution model and accepts any distribution policy. Distribution is a major issue in complex systems, however, and should be taken into account and specified during software development. In this section, we study distribution as a separate concern.

To specify distribution, we need to specify physical location of processes and data objects. Hence, we assume a set R containing rule names and a set T containing data types. Data types are used to categorize data items used/produced by different rules. To simplify the following discussion, we assume that data objects of each type are present only at one site at each point of time. We do not specify how to assign this typing to variables but assume that there is a function from the set of variables to types ($type : Var \rightarrow T$), returning the type of a variable. The set of physical locations is denoted by P. Static distribution is defined as a function $StaticDist : R \cup T \rightarrow P$, representing the physical locations of the data objects of each type and rules. Note that we did not restrict locations to contain both data and processing (rules) and hence, a physical location may represent a storage node or processing unit, or both.

This general specification of distribution can be used to model more specific distribution policies, such as push and pull models. For example in a push model, the function *StaticDist* should map any data type to its consumer side. In a pull model, however, the data type remains on the producer side and should be accessed (fetched) from the producer by the consumer.

We should note that preventing inconsistencies in accessing shared data items is still provided in the basic GAMMA semantics and need not be considered here. Nevertheless, if an application calls for its own dedicated consistency control algorithm, it should be specified in the form of stronger conditions in rules or an extension of the GAMMA model to a new aspect (by defining a notion of (in)dependence for parallel execution).

Example 3 *Distribution in our elevator system.* Suppose that sensors for

request buttons on each floor are connected to the elevator via a fieldbus network. In this case, accessing the distributed nodes will take some time from the elevator. To specify this model of distribution, we assume a physical node for the elevator and its internal buttons and a node for each external button. The distribution function for the elevator system then looks like the following:

$$StaticDist(type((extStop, i), status)) = floor_i$$

$$StaticDist(type((inStop, i), status)) = elevator$$

$$StaticDist(type(cf, i)) = elevator$$

for each rule $rule$: $StaticDist(rule) = elevator$

This distribution policy defines where the GAMMA rules *moveDown* and *moveUp* must look for remote copies of external request values from distributed nodes.

This distribution model should be further combined with the functionality and timing model in one semantics in order to verify that the system satisfies properties that depend on the combination of several aspects.

3 Weaving Aspects

The idea of weaving is composing different aspects of design. In our case, we have to relate functionality, (coordination,) timing, and distribution, and present them in one semantic model. The orthogonality of non-functional aspects allows the designer of each aspect to neglect the other. As a result, the weaving process reflects change or even absence of one aspect in the whole semantics.

A GAMMA specification presents functionality in the form of independent rules. The timing specification aspect relates a rule to an interval representing execution time. The distribution aspect defines physical distribution of rules and data items.

If there is no timing estimation specified for a rule (as it is the general case for un-timed specifications), it is assumed to be $[0, \infty]$, i.e., an arbitrary execution time. If the distribution aspect is absent, a single location is assumed. Our proposal for a formal semantics of weaving consists of a timed transition system [10] with transitions of a GAMMA program and timing consisting of computation time plus communication time.

We denote the computation time of a rule r by $comp(r)$. As mentioned before, if there is no interval defined for a rule r, $comp(r)$ results in $[0, \infty]$. This function induces a *by-name* weaving method to relate GAMMA rules and their respective timing estimations. In this paper, we assume that $comp(r)$ works as a function returning the execution time estimation of a rule, if available, or otherwise $[0, \infty]$. Nevertheless, this assumption could be relaxed by allowing several intervals associated to a rule, and hence letting $comp(r)$ return one of the intervals non-deterministically (or a set of intervals). This could be used to model the situation where a rule has multiple possible execution times, depending e.g. on varying implementation environments.

To represent communication costs resulting from the distribution policy, we use the function $comm(r)$, which returns the time cost for making local copies of the data items needed for the execution of rule r. For a rule r, $comm(r)$ is computed by taking the maximum of communication costs for all variables (of data items) v present in rule r, that reside in a different physical node than r ($StaticDist(type(v)) \neq StaticDist(r)$). If all the data needed for the execution of a rule is available at the location of the rule itself, we assume the communication cost to be 0.

Example 4 *Weaving of aspects of our elevator system.* In Figure 3, a fragment of the timed transition system is given that results from weaving the computation, timing and distribution of this system as described in previous examples. The transitions are labelled by the name of the rule(s) that are executed, the timing estimation of the execution, and the communication cost. For simplicity, only the relevant elements of the multiset contents are represented in this figure. It is assumed that the time cost for communicating data from one node to another is CT.

The simple time weaving function presented here can be extended by adding estimations for failed attempts to execute a rule, or by defining the timing estimation as a function of multiset size or contents. In GAMMA, rule implementations, computation time and failure time may depend on the time for searching the multiset to find the appropriate valuation. These two extensions

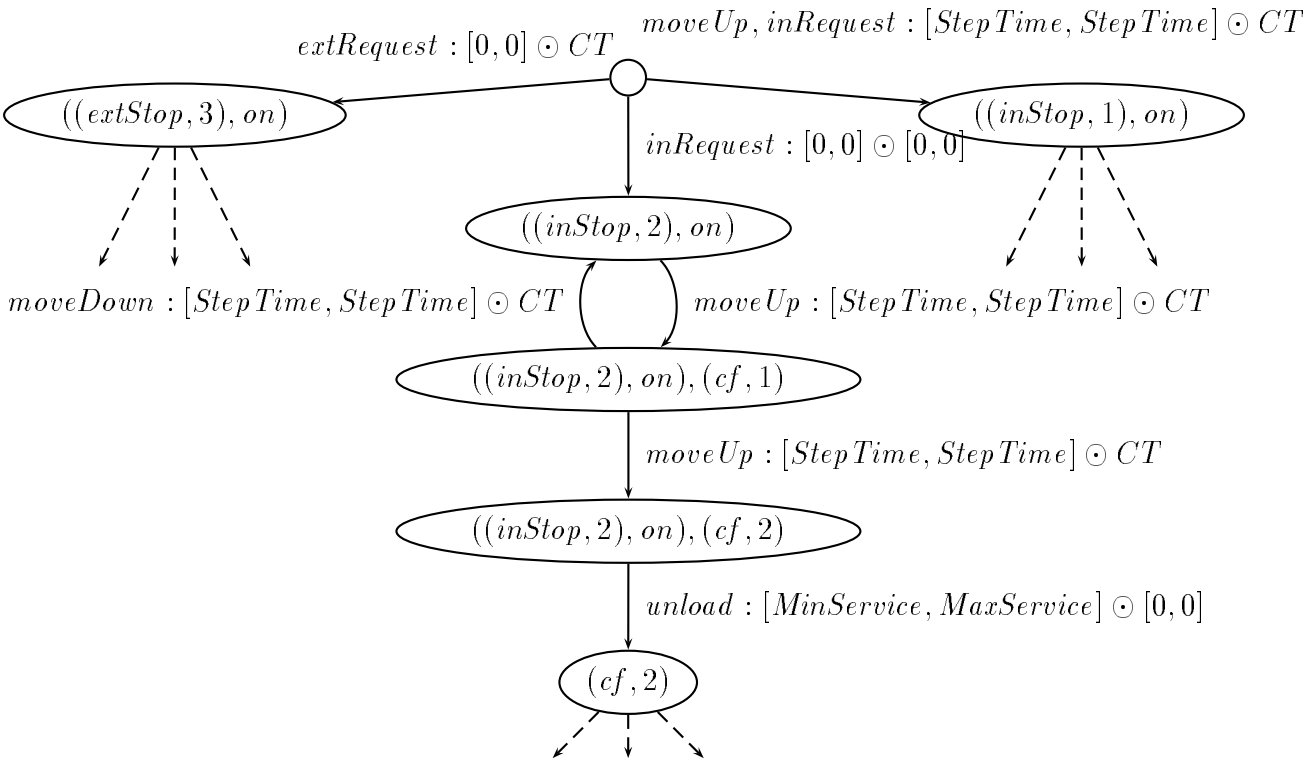

Fig. 3. Fragment of the Timed Transition System after Weaving.

thus add to the practical value of the proposed method. Such extensions can illustrate the profit of the separation of concerns in the method outlined in this paper.

4 Related Work

A process algebraic framework is an appropriate choice for formalizing aspects. In [1], process algebra is suggested as a formal framework for aspect-oriented design. Although we acknowledge the importance of a process algebraic (and in general a formal) framework in establishing a semantics for aspect-orientation, we believe that a single process algebra will be most useful in modeling a *weaved* specification. In other words, by using a process expression for different aspects of modeling, the problem of entangling different aspects in a single expression cannot be solved completely.

In [7], an extension of the GAMMA formalism (namely Structured-GAMMA) is used as a specification language for an aspect-oriented implementation of a distributed shared memory protocol. There, the authors mention the benefits of abstraction from distribution in the GAMMA programs and the possibility of formal reasoning using this specification language.

5 Conclusion and Future Research

The current trends in AOP [6] can be summarized as follows:

(1) Semantic correctness of aspects and compositions.
(2) Defining methods for identifying and specifying canonical models for crosscutting concerns, including methods for composing aspect models.
(3) Defining formal models for determining functional and quality characteristics of crosscutting concerns individually and together.

We can summarize our contribution to these challenges as follows:

(1) We provided some ideas for the formal design of a small number of aspects, mainly related to distributed real-time systems, which are kept separate and abstract from each other.
(2) These aspects are weaved together into a single semantic framework.

The main challenges in our future research are the following:

- Providing a formal syntax, weaving, and semantics of the aspects discussed in this paper. In [12], a formal syntax and semantics are given for basic functionality, coordination, and timing aspects.
- Extension of the method sketched in this paper to other aspects such as coordination, power-awareness, fault-tolerance, persistency, etc.
- Developing/studying logics for expressing properties of the aspect models and the weavings of those.
- Performing case studies to validate the method.
- Developing automated design methods and tools that support the aspect weaving process and the reasoning in the aspect models.

References

[1] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209, Berlin, 2001. Springer-Verlag.

[2] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In Cristian S. Calude, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer-Verlag, Berlin, 2001.

[3] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM (CACM)*, 36(1):98–111, January 1993.

[4] Michel R. V. Chaudron. Separation of Correctness and Complexity in Algorithm Design. Technical Report 94-36, Leiden, The Netherlands, 1994.

[5] Michel R. V. Chaudron. *Separating Computation and Coordination in the Design of Parallel and Distributed Programs.* PhD thesis, Department of Computer Science, Rijksuniversiteit Leiden, Leiden, The Netherlands, 1998.

[6] Tzilla Elrad, Robert E. Filman, and Atef Bader. Special issue on aspect oriented programming. In *Communications of ACM.* ACM Press, 2001.

[7] P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2-3), December 2000.

[8] Graeme Smith. *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer, Boston, first edition, 2000.

[9] Chris L. Hankin, Daniel Le Métayer, and David Sands. A calculus of Gamma programs. In *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Machines*, volume 757 of *Lecture Notes in Computer Science*, pages 342–355, Berlin, 1993. Springer-Verlag.

[10] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In Jaco W. de Bakker, Kees Huizing, Willelm P. de Roever, and Grzegorz Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 226–251, Berlin, 1992. Springer-Verlag.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[12] M.R. Mousavi, T. Basten, M.A. Reniers, M.R.V. Chaudron, and G. Russello. Separating functionality, behaviour and time in the design of reactive systems: (GAMMA + coordination) + time. 2002.

[13] P. Tarr and H. Ossher and W. Harrison and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, May 1999.

[14] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[15] AspectJ Website. http://www.aspectj.org.