Nordic Journal of Computing 15(2014), 75–111.

TIMED-GAMMA AND ITS COORDINATION LANGUAGE

MOHAMMAD REZA MOUSAVI TWAN BASTEN
MICHEL A. RENIERS
Eindhoven University of Technology (TU/e)
The Netherlands
M.R.Mousavi@tue.nl, A.A.Basten@tue.nl, M.A.Reniers@tue.nl

MICHEL CHAUDRON
Leiden Institute of Advanced Computer Science
The Netherlands
chaudron@liacs.nl

Abstract. This paper proposes a theoretical framework for *separation of concerns* in the formal specification of reactive and real-time systems. This framework consists of the syntax and the semantics of three languages (and all meaningful combinations thereof) that each address a separate concern. The first language, *Gamma* (a variant of an existing language) is used to define the functionality of a system (by means of a set of basic data transformations). Our additions are a simple language of intervals specifying timing-properties of basic transformations and a language (called *Schedules*) for specifying the coordination of the basic *Gamma* transformations. Each of these languages formally models a separate aspect of a system and statements in these languages can be reused, changed or analyzed in their own right. Our key contribution is that we provide a formal framework in which different combinations of aspects have a well-defined semantics.

ACM CCS Categories and Subject Descriptors: F.3 [Logics and Meanings of Programs]; D.3.1 [Formal Definitions and Theory]; D.1.3 [Concurrent Programming]

Key words: separation of concerns, reactive systems, coordination, real-time systems, shared data-space, formal methods

1. Introduction

1.1 Motivation

Separation of concerns in software design has been proposed in several classic computer science texts since the very beginning of this discipline [Dijkstra 1976]. A major motivation for advocating this principle has been to conquer the complexity of designing software by means of breaking it into a number of design issues, aspects or concerns (e.g., timing, scheduling, persistency or security) that can be addressed separately [Elrad *et al.* 2001, Tarr *et al.* 1999]. Hence, providing abstract and simple formalisms that are tailor-made for a single concern of requirement specification, design, or programming, is of great importance. Tailor-made

Received February 8, 2009; accepted May 24, 2010. Communicated by Kim G. Larsen.

formalisms can enable a more focused design method that allows designers to concentrate on each aspect of a design separately. Furthermore, they ease changing an aspect without being directly involved with other ones. Also, separation of concerns facilitates reuse of each aspect in other designs. Recently, a renewed interest appeared in separating different concerns and providing appropriate ways of focusing on each concern. A distinguished example of this trend can be seen in Post Object Oriented Programming languages (POPs) [Elrad *et al.* 2001;Harrison and Ossher 1993] and in particular in the Aspect Oriented Programming (AOP) [Elrad *et al.* 2001] and Multi-Dimensional Separation of Concerns [Tarr *et al.* 1999] methods.

The ultimate goal of the research commenced by this paper is to have a set of declarative and abstract specification languages for each aspect of a system (e.g., functionality, timing, distribution, coordination, etc.). Any meaningful combination of aspect designs can then be weaved together to reflect inter-connections of aspects. Different aspect models (specifications of different aspects) can be refined in order to move towards a more restricted model (and ultimately towards an implementation; in this paper, however, we are only concerned with modeling and do not consider implementation issues). System properties can be deduced from individual aspects with proven properties. For example, if some class of properties is only dependent on the functionality aspect (i.e., it only refers to basic reactions or input-output behavior and weaving other aspects such as timing and scheduling preserves the properties), then we can simplify the procedure of verifying these properties by only examining the semantics of the functionality model. After a correctness analysis of the set of designs, an executable behavioral model can be derived from it. For validating non-functional aspects (i.e., aspects not concerned with the pure input-output or reactive behavior such as timing), a set of monitoring components might be generated from a specification so that these aspects of specification can be checked at run-time, too.

This paper takes a step toward this ultimate goal by providing a formal framework of separation of concerns for real-time systems using the shared-data-space metaphor. This metaphor prescribes that the independent (possibly parallel) components performing basic functionality do not carry any explicit reference to one another and instead only communicate via a shared data-space. Fig. 1 shows a schematic view of the proposed method. The novelty of this work compared to the approaches mentioned above is that, first, it exploits the idea of separation of concerns at the specification and design level, and, second, it establishes a robust theoretical basis that allows rigid analysis and verification of (timed) designs. This is facilitated by having separate, yet composable, semantics for the different concerns/aspects.

1.2 Summary of approach

In our approach, the basic functionality of a system is designed using an abstract formal model of computation called *Gamma* [Banâtre *et al.* 2001]. A *Gamma* program consists of a set of basic data transformations called *rules*. Timing information is associated to *Gamma* rules in the form of separate intervals. The resulting

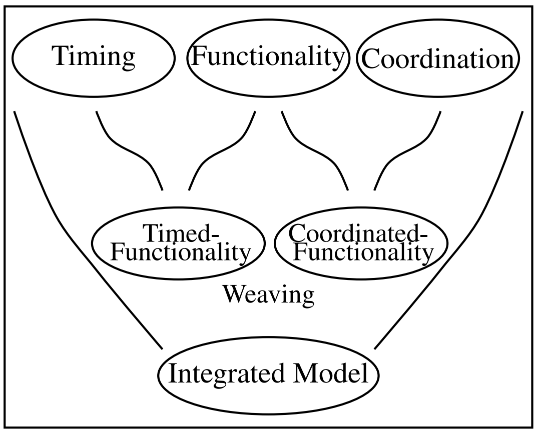

Fig. 1: Separation of concerns in design.

timed-functionality modeling language is called *Timed-Gamma*. Composed behavior of the system is expressed in a coordination language named *Schedules*, specifying the order of (timed) computations using constructs such as sequential composition and parallel composition. More details follow in Section 4.

1.3 Related work

Separation of computation and coordination has been extensively investigated in the area of coordination languages (for an overview, see [Papadopoulos and Arbab 1998;Gelernter and Carriero 1992]). These languages can be generally classified into two main categories: data- and control-driven languages [Papadopoulos and Arbab 1998].

In data-driven coordination languages [Brogi and Jacquet 2003], the main goal is to abstract from the communication mechanisms by providing a shared data-space as the communication medium among components. This allows for spatial and temporal decoupling of components and provides a level of separation of concerns. Linda [Carriero and Gelernter 1989] and *Gamma* [Banâtre *et al.* 2001] are well-known examples of such languages.

On the other hand, control-driven coordination languages provide the possibility of defining exogenous control strategies over a set of components. In other words, these languages provide the primitives to compose basic functionality units (e.g., simple reactions producing some output as a result of a certain input) as black-box components. Manifold [Arbab 1996], ToolBus [de Jong and Klint 2003] and Synchronizers [Frølund and Agha 1993] are examples of control-driven coordination languages / architectures.

In our approach, we try to benefit from the merits of both of these categories. First, we lay a layer of abstraction in our communication platform (i.e., the way basic functionality units are composed) by using the shared-data-space metaphor of *Gamma*. Using *Gamma* one can define basic functionality units producing a certain output (e.g., a tuple) in case a certain input (again in the form of a tuple) is present in the shared data-space. Then, we introduce our *Schedules* language to specify control and coordination strategies of the basic functionality units, following the ideas of Chaudron and de Jong [1996, 1998]. Furthermore, by extending this paradigm to timed settings, we allow for timing analysis of both functionality and coordinated behavior models.

With respect to timed-coordination languages, a few attempts have been made to extend coordination languages with time (such as [Papadopoulos and Arbab 1996; Bergstra and Klint 1998; Nielsen *et al.* 1998; Jacquet *et al.* 2000; Hannemann and Hooman 2001; Arbab and Rutten 2003; Arbab *et al.* 2004; Hooman and van de Pol 2005]). In the area of data-driven coordination languages, Jacquet *et al.* [2000] presents four different timed extensions of the concurrent Linda coordination language (namely, with relative delay, i.e., delay after finishing the last task, absolute wait, i.e., waiting till a certain moment of time, relative durational primitives, i.e., primitives that take a certain amount of time before their execution can or must be completed and absolute durational primitives, i.e., primitives that have to delay till a certain moment of time). Although relative durational primitives are conceptually close to our extension of *Gamma* rules, they differ from ours in a number of ways. First, they force time-stamping the data-space and only allow introduction and consumption of temporary elements to/from the data-space. Second, *Gamma* has a built-in transaction-based mechanism (in terms of rules) which is absent in Linda. Furthermore, time transitions are synchronized among all durational primitives in [Jacquet *et al.* 2000] while we allow for both synchrony and asynchrony of time transitions.

Also in [Hannemann and Hooman 2001; Hooman and van de Pol 2005] a framework is proposed for compositional reasoning about real-time shared-data-space systems in PVS. In this approach, data elements are time stamped when introduced in the shared data-space. We do not require time-stamping in our framework since it is not necessary when only timing information of basic transformations suffices for the specification of a system. Nevertheless, one can implement time-stamping and temporary (temporal) data items in our framework.

For the purpose of control-driven coordination languages, in [Papadopoulos and Arbab 1996], Manifold is used to coordinate real-time components. The main differences between our work and [Papadopoulos and Arbab 1996] arise from the fact that they assume timing is described in the functionality of components (or components are instrumented with timing measures and checks). This leads to the fact that components themselves can take care of timing by requesting, releasing and passing control. However, we try to separate timing concerns from functionality of components so that real-time behavior can be reflected automatically in the semantics and can be implemented independently from components using monitors or observers. Reo [Arbab and Rutten 2003], a successor of the coordination language Manifold, uses a qualitative notion of time. The notion of time there represents the causal ordering between communication actions and is very close to the notion of tags in the tagged signal model of [Lee and Sangiovanni-Vincentelli 1998].

The ToolBus coordination architecture is also extended to discrete timed settings in [Bergstra and Klint 1998]. In the Discrete-Time ToolBus, the user can choose between absolute- and relative-time versions of delay and time-out primitives. These primitives are controlled by the central clock of the ToolBus. The focus in the Discrete-Time ToolBus approach is more on the coordination aspect (by using so called term scripts). However, in our approach we want to address aspects of functionality and timing as separate aspects of design. Thus, in the Discrete-Time ToolBus, functionality of components (tools) are abstracted (treated as black-box) and timing primitives are added to the coordination language.

RTSynchronizers [Nielsen *et al.* 1998] is the real-time extension of Synchronizers [Frølund 1992; Frølund and Agha 1993]. In RTSynchronizers the behavior of an actor-based system is restricted by specifying real-time constraints over communication (method invocation) patterns. To apply RTSynchronizers to untimed actor models, arbitrary time delay (time-pass) for each action execution is assumed. Then the (chaotic) actor execution patterns are restricted using a superimposition of RTSynchronizers. Intuitively, the work of Nielsen *et al.* [1998] is in the same direction as the approach of this paper. However, there are a number of essential differences. Firstly, Nielsen *et al.* [1998] do not allow for any assumption about timing of basic actions. However, in our approach we allow for both unrestricted timed behavior of rules as well as explicitly added timing performance specifications to them. Secondly, an RTSynchronizer specifies a mixture of real-time and coordination requirements (e.g., by mixing constructs for causal ordering and deadlines) and thus is not entirely in line with our design philosophy of separation of concerns.

There are also several extensions of process calculi with timing in the literature. Overviews of such extensions can be found in [Vereijken 1997; Davies and Schneider 1995]. Our *Schedules* languages (considering its timed semantics) can be categorized as a process calculus with relative intervals and delayable actions. In most timed-process algebras, atomic actions are treated uniformly (as always being enabled to execute). Also, timed-synchrony is the dominant choice among timed-process algebras which means that all parallel components should be executed in true concurrency (in time). We do not have this uniformity assumption in our rules, since enabledness is defined based on the contents of the shared data-space. Furthermore, we allow a kind of asynchronous timed parallelism in which actions can be both serialized (preempted by other actions) and executed in true concurrency. In our view, keeping timing and resource availability concerns separate from causal relations of actions (thus, timed asynchrony) is essential, especially in early stages of the design where the details of the actual implementation domain are not known. There has been a similar attempt to define an abstract notion of timed parallel composition in [Aceto and Murphy 1996]. In this process algebra, non-synchronizing actions are forced to make asynchronous (interleaving) time transitions and synchronizing actions are specified to perform synchronous (concurrent) time transitions. This distinction is not necessary in our framework, since we do not have explicit synchronization points.

In [Mousavi *et al.* 2003], we used our approach to model the well-known steam-boiler control case-study of Abrial *et al.* [1996]. Among the attempts to formalize the steam-boiler case study, those using process algebraic formalisms are close to our approach. For example, Willig and Schieferdecker [1996] use Time Extended

Lotos to formalize the steam boiler case. An interesting observation about the case study is that the specifiers were forced to use another language (a functional programming language) to specify their functionality aspect. This shows that process algebraic approaches are more suitable for specification of behavior/coordination and timing (usually in a mixed fashion) than for the specification of functionality. Our method tries to provide an integral approach that supports separate aspect specifications.

1.4 Case study: requirements

To illustrate different aspects of our method, we treat a simplified version of the light control system design [Börger and Gotzhein 2000]. Our light control system is built upon a sensor network that is used for different purposes in a smart building. We abstract from the sensor layer specification and assume that the sensors give accurate and in-bound information about the positions of individuals inside the building, based on some fixed system of coordinates. The building is rectangular and divided into rectangular rooms with defined boundaries. Individuals may move within the building with a certain speed and leave or enter the building through a number of doors.

A control strategy should be devised such that if an individual is detected in a room, the lights of that room should be switched on. If a person has just left the room and there is no other person in the room, then lights should turn off. The goal of the case study is to guarantee that whenever a person is in a room the lights of that room will turn on within a certain time boundary. Also when the room is empty, the lights should be turned off within a boundary. We first model the functionality of the system, then add timing information to it and finally we devise a coordination (control) strategy to meet the requirements.

1.5 Structure of the paper

The rest of this paper is organized as follows. Section 2 introduces our data and functionality model as well as the preliminary mathematical concepts behind them. Section 3 introduces the timing aspect and also defines how the functionality model and the timing aspect can be composed. Section 4 presents our coordination language for ordering the behavior of (*Timed-*)*Gamma* programs. Subsequently, Section 5 defines equivalence notions among timed-programs and schedules that are essential in compositional reasoning and refinement. Section 6 defines a schedule that represents the chaotic behaviour of a *Gamma* program. Finally, Section 7 concludes the results and shows directions of our future research.

2. Functionality

2.1 Motivation: Shared-Data-Space paradigm

The Shared-Data-Space paradigm is a metaphor for component interaction that allows for temporal and spatial decoupling of components. As shown in Fig. 2, com-

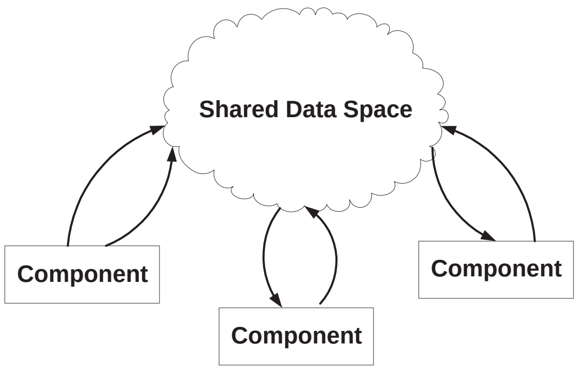

Fig. 2: Shared-Data-Space model.

ponents can access the data-space as the shared communication medium independent from each other. *Gamma* (**G**eneral **A**bstract **M**odel for **M**ultiset **Ma**nipulation) [Banâtre and Le Métayer 1993; Banâtre *et al.* 2001] is a specification/programming model based on this paradigm. In *Gamma*, the shared data-space is modeled by the notion of a *multiset* (bag) and functionality of components is represented by *rules* which model basic transformations on multisets.

Gamma rules are specified independently from each other and they do not make reference to each other. All possible interactions among basic functionality units (e.g., ordering or synchronization) are thus abstracted away through the notion of shared data-space. This also means that rules can be applied in any order to the shared data-space using all possible levels of true concurrency. As a consequence, unneeded sequentiality (e.g., immense use of semi-colon notation even in parallel versions of classic programming languages [Dijkstra 1976]) is not imposed on *Gamma* programs. In other words, *Gamma* programs can capture the parallelism that is logically inherent in the problem definition [Banâtre and Le Métayer 1993].

This abstract nature of *Gamma* makes it a suitable choice for the specification of basic functionality units of software components. Using this model allows component designers to concentrate on basic units of functionality and leave the composed behaviors as well as other non-functional aspects of them to be devised in later design phases, and/or by other specification methods (e.g., coordination languages for specifying composed behavior). A *Gamma* program encompasses every arbitrary ordering of basic actions with arbitrary levels of true concurrency.

In the remainder of this section, we first introduce the notion of multiset as a mathematical model for a shared data-space. Then, we define substitutions and computations on multisets (as the semantic framework for execution of rules) and investigate their properties. Consequently, we give the formal syntax of *Gamma* programs and elaborate on the intuition behind them. Finally, we specify the functionality aspect of the case-study. We defer giving a complete semantics to *Gamma* programs until the next section where we introduce the timing aspect. There, we present an un-timed interpretation of *Timed-Gamma* programs. The only reason for doing so is to prevent the redundancy caused by presenting two similar semantics. However, elsewhere, we have defined a purely untimed semantics for our version of *Gamma* programs [Mousavi *et al.* 2002], which is reasonably different from the original semantics of [Banâtre and Le Métayer 1993] due to the presence of true concurrency. Our true concurrency semantics of *Gamma* [Mousavi *et al.* 2002] (in particular the notions of concurrent computation and independence in this semantics) are especially important for our move to the timed setting.

2.2 Multisets and computations

2.2.1 Basic definitions

In this subsection, we define a concise and basic theory of multisets that serves as a mathematical meaning for the concept of a shared data-space in *Gamma*. For a more detailed discussion on multisets and some historical accounts, see [Syropoulos 2001].

Definition 1. (Multiset) *A multiset is a set that allows multiple occurrences of elements. More precisely, it is defined as a total function from a set U (for universe) of elements to the set of natural numbers $\mathbb{N}$ representing their number of occurrence. Hence, a multiset M is defined as:*

$$M : U \to \mathbb{N}.$$

We refer to the set of all multisets of a universe U as $\mathbb{M}(U)$. The empty multiset is denoted by $\varnothing$. We use the notation $e \sqsubset\!\!\!\!- M$ to denote membership, $[e_0, e_1, \ldots]$ for external representation and $M_0 \sqsubseteq M_1$ for multisubset. Addition and subtraction of multisets are denoted by $\boxplus$ and $\boxminus$, respectively. Union and intersection of multisets (taking the maximum and minimum number of occurrences of each element) are denoted by $\sqcup$ and $\sqcap$, respectively. Elements of our multisets are closed terms built upon a predefined logical structure (a first order language) and are closed under pairing. Given a fixed set of variables, the set of open terms from this structure closed under pairing is called the set of basic expressions.

Example 1. Consider the case-study requirements given in Section 1.4. To design a multiset for this case study, we assume a basic set B that contains the integers. Associated with this set are the usual basic operators such as addition and the usual relations such as equality and ordering. The multiset for this case study is defined to contain elements of the following types:

(1) The architecture of the building (floorplan): The boundaries of the building are denoted by a tuple $(\text{BuildingDim}, x_1, y_1, x_2, y_2)$. This means that the rectangular building is placed on a two-dimensional grid with lower-left corner at position (x_1, y_1) and upper-right corner at position (x_2, y_2). The boundaries of the room are denoted by tuples of the form $(\text{RoomDim}, i, x_1, y_1, x_2, y_2)$. Doors are denoted by tuples (Door, x, y) showing their coordinates. It is assumed that the rooms do not overlap and cover the whole building. Also all doors are on room boundaries. It is assumed (without checking) that any initial multiset respects these spatial restrictions.

(2) Individuals: Ordinary persons are denoted by tuples of the form (Person, i, x, y) where Person is a name, i is the identifier of the person, and the pair x, y shows the person's coordinates (tuples are terms that are constructed using pairing). When a person moves out of the building, this will be denoted by (PersonOut, i).

(3) Lights and sensors: We assume that there exists a tuple representing the lights of each room, denoted by (Light, i, *status*) where i is the room number and *status* can be either On or Off. The occupancy status of room i is indicated by tuples of the form (RoomStat, i, *status*) where status can be Occ if the room is occupied or Free otherwise.

2.2.2 Substitution, computation and independence

The basic notion of computation in *Gamma* is the rewriting of a multiset. The rewriting of a multiset due to one rule is modeled by a substitution. A computation consists of an arbitrary number of substitutions. This way, a multiset of (simultaneous) substitutions models a parallel execution of rules. Note that usually computation is defined in terms of a sequence of basic steps; our notion of computation, however, admits simultaneous steps and the ordering of our (simultaneous) computation steps is later defined in the semantic frameworks to be presented in the forthcoming sections. This is to admit all possible orderings that can be prescribed by the coordination specification.

Definition 2. (Substitution and Computation) *For multisets N and N', the expression N/N' is called a* substitution *of N for N' (typically denoted by α, α_1, etc.). A* computation *is a multiset of substitutions (typically denoted by σ, σ_1, etc.).*

Intuitively, applying a computation with a single substitution $\sigma = [N/N']$ to multiset M should result in taking multiset N' from M and putting back multiset N. In this operation, some parts of the multiset may be only taken away temporarily by N' and put back by N again. We call this the read part following the intuition that this part is not really taken away but only read. The parts that are permanently removed and added by a substitution are called take and put parts, respectively. We lift this intuition to general computations, as follows. The *read* part of a computation is the union of read parts of its substitutions because a single copy of an element can be read by several substitutions concurrently. For the *take* or *put* parts, different copies of the elements are removed or added by the individual substitutions.

Definition 3. (Computation: Parts and Application) Read*,* take *and* put *parts of a computation are defined as follows:*

$$\begin{aligned}
&\text{read}(\varnothing) \triangleq \text{put}(\varnothing) \triangleq \text{take}(\varnothing) \triangleq \varnothing\\
&\text{read}([N/N'] \boxplus \sigma) \triangleq (N \sqcap N') \sqcup \text{read}(\sigma)\\
&\text{take}([N/N'] \boxplus \sigma) \triangleq (N' \boxminus \text{read}([N/N'])) \boxplus \text{take}(\sigma)\\
&\text{put}([N/N'] \boxplus \sigma) \triangleq (N \boxminus \text{read}([N/N'])) \boxplus \text{put}(\sigma).
\end{aligned}$$

Application of a computation σ to a multiset M is defined as:

$$M(\sigma) \triangleq \begin{cases} (M \boxminus \mathrm{take}(\sigma)) \boxplus \mathrm{put}(\sigma) & \textit{if } \mathrm{read}(\sigma) \boxplus \mathrm{take}(\sigma) \sqsubseteq M \\ M & \textit{otherwise.} \end{cases}$$

If read(σ) $\boxplus$ take(σ) $\sqsubseteq M$*, then we call* σ consistent *with respect to* M*. We also denote the fact that* σ *is consistent with respect to* M *by* $M \models \sigma$*.*

In order to model parallelism in our *Gamma* semantics, we introduce a notion of independence on computations. Two computations are independent if both can be applied simultaneously or in an arbitrary order.

DEFINITION 4. (INDEPENDENCE) *Two computations* σ_0 *and* σ_1 *are* independent *with respect to a multiset* M*, denoted by* $M \models \sigma_0 \bowtie \sigma_1$*, if and only if* read($\sigma_0 \boxplus \sigma_1$) $\boxplus$ take($\sigma_0 \boxplus \sigma_1$) $\sqsubseteq M$*. As an alternative (but formally equal) definition* $M \models \sigma_0 \bowtie \sigma_1$ *if and only if* $\sigma_0 \boxplus \sigma_1$ *is consistent with respect to* M*:* $M \models \sigma_0 \boxplus \sigma_1$*.*

As a consequence, two computations are independent if and only if both can find enough shared copies of elements to read and enough different copies of elements to take.

EXAMPLE 2. Consider the substitution

$$\alpha_0 = [(\mathrm{PersonOut}, 1), (\mathrm{Door}, 10, 10)]/[(\mathrm{Person}, 1, 10, 10), (\mathrm{Door}, 10, 10)].$$

This represents person number 1 leaving the building. In this substitution, the tuple representing the position of Door is only read. However, the tuples representing person 1 inside and outside the building are taken and put, respectively. Suppose that we have another substitution

$$\alpha_1 = [(\mathrm{Person}, 2, 10, 10), (\mathrm{Door}, 10, 10)]/[(\mathrm{PersonOut}, 2), (\mathrm{Door}, 10, 10)]$$

which represents person number 2 entering the building using the same door. If our initial multiset is $M = [(\mathrm{Person}, 1, 10, 10), (\mathrm{Door}, 10, 10), (\mathrm{PersonOut}, 2)]$, then computations $[\alpha_0]$ and $[\alpha_1]$ are independent with respect to M. Hence, they can be applied to M simultaneously or in any arbitrary order and in any case this results in the multiset $[(\mathrm{Person}, 2, 10, 10), (\mathrm{Door}, 10, 10), (\mathrm{PersonOut}, 1)]$.

The original presentations of the *Gamma* formalism [Banâtre and Le Métayer 1993; Hankin *et al.* 1993] do not have a notion of concurrent computation and hence neither of the concept of independence. Our notion of independence is more liberal than the notion of [Chaudron 1998], in that it allows for more concurrent tasks to execute due to separating read and take parts (i.e., it allows for sharing the data items that are only read). Apart from the theoretical differences among the resulting untimed semantics, the notions of concurrent computation and independence are of essential importance in defining the semantics for *Timed-Gamma*.

2.3 Gamma syntax

The abstract syntax of basic *Gamma* rules is presented in Fig. 3. A *Gamma* program consists of a name (a string of syntactic class `ProgramName`), and a non-empty set of rules, each rewriting the content of the shared multiset. Each rule consists of a name (represented by the syntactic class `RuleName`) and a set of terms at the left- and right-hand side of the substitution arrow $\mapsto$ and a condition part that are to be valuated by the multiset content. In the given syntax, `BasicExp` and `Condition` stand for open terms and logical formulae built upon a given logical structure on multiset elements (see Definition 1). Either of the multiset expressions in the right or left side of a rule can be empty (represented by the empty string symbol ϵ).

```
Program      ::=  ProgramName = {Rules}
Rules        ::=  Rule | Rule, Rules
Rule         ::=  RuleName = MultisetExp ↦ MultisetExp
                  ⇐ Condition
MultisetExp  ::=  ϵ | MExp
MExp         ::=  BasicExp | BasicExp, MExp
```

Fig. 3: Basic *Gamma* syntax.

In some *Gamma* rules, there is a need to read a term from the multiset in order to check some conditions on its value, to use its value for substitution of other elements, or just to check the existence of it. To do this in basic *Gamma* syntax, one has to mention the tuple in both right- and left-hand side of the rule. To represent this notion more concisely, we define the following syntactic shorthand:

Definition 5. (Explicit read) *For an arbitrary deduction rule rn, when an expression e is only read from the multiset and no manipulation on it is required, then we denote this by* $rn = e? : \ lhs \mapsto rhs \Leftarrow c$*, which is defined to be* $rn = e, lhs \mapsto e, rhs \Leftarrow c$*. Similarly, a multiset expression (a list of basic expressions) can be read using the* $?$*-notation after each basic term.*

Example 3. The following *Gamma* program represents simple rules, explained below, modeling movement of people inside the building and their ability to enter and leave the building:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Movement} =} \\
\{\ \text{Move} & = & (\text{RoomDim}, k, x_1, y_1, x_2, y_2)? \\
 & & : (\text{Person}, i, x, y) \mapsto (\text{Person}, i, x + x', y + y') \\
 & & \Leftarrow x'^2 + y'^2 \leq \text{Speed}^2 \wedge \\
 & & \quad x_1 \leq x \leq x_2 \wedge y_1 \leq y \leq y_2 \wedge \\
 & & \quad x_1 < x + x' < x_2 \wedge y_1 < y + y' < y_2, \\
\quad \text{Move}_{\text{d}} & = & (\text{RoomDim}, k, x_1, y_1, x_2, y_2)?, (\text{Door}, x + x', y + y')? \\
 & & : (\text{Person}, i, x, y) \mapsto (\text{Person}, i, x + x', y + y') \\
 & & \Leftarrow x'^2 + y'^2 \leq \text{Speed}^2 \wedge \\
 & & \quad x_1 \leq x \leq x_2 \wedge y_1 \leq y \leq y_2 \wedge \\
 & & \quad x_1 \leq x + x' \leq x_2 \wedge y_1 \leq y + y' \leq y_2, \\
\quad \text{MoveIn} & = & (\text{BuildingDim}, x_1, y_1, x_2, y_2)?, (\text{Door}, x, y)? \\
 & & : (\text{PersonOut}, i) \mapsto (\text{Person}, i, x, y) \Leftarrow \text{Entrance?}, \\
\quad \text{MoveOut} & = & (\text{BuildingDim}, x_1, y_1, x_2, y_2)?, (\text{Door}, x, y)? \\
 & & : (\text{Person}, i, x, y) \mapsto (\text{PersonOut}, i) \Leftarrow \text{Entrance?} \\
\} & &
\end{array}
$$

where Entrance? denotes the predicate

$$((x = x_1 \vee x = x_2) \wedge y_1 \leq y \leq y_2) \vee (x_1 \leq x \leq x_2 \wedge (y = y_1 \vee y = y_2)),$$

i.e., the door involved is an outer door of the building.

Rule Move specifies that coordinates of a person may change due to the movement by at most its speed limit (Speed: a constant) for each step. Meanwhile the firm boundaries of the room need to be respected. The second rule describes that, under similar restrictions, a person may move to a door (of this room). The rule MoveIn specifies that a person residing outside the building can always step in at an entrance. Finally, rule MoveOut specifies that a person can leave the building if s/he is standing at a door.

These four rules model the functionality of the underlying sensor network as outlined in the case-study requirements.

Due to our aim of separation of concerns, the presented syntax of *Gamma* is simpler than the original one [Banâtre and Le Métayer 1993]. It contains only the basic functionality (rule) part of programs and postpones all structuring and control decisions to defining an appropriate coordination strategy for a program. Hence, structuring techniques like tropes in [Hankin *et al.* 1993] and composition operators in [Chaudron 1998] are omitted.

EXAMPLE 4. (LIGHT CONTROL: FUNCTIONALITY) In this example, we present a functionality model for the light-control case study. To manage the complexity of the functionality model itself, we break the functionality model into two *Gamma* programs. The first program is the program named Movement as given in Example 3. The second program defines the functionality of the sensors and the central control module.

$$
\begin{array}{lll}
\text{Control} = & & \\
\{\ \text{RoomOcc} & = & (\text{RoomDim}, k, x_1, y_1, x_2, y_2)?, (\text{Person}, i, x, y)? \\
 & & : (\text{RoomStat}, k, \mathit{status}) \mapsto (\text{RoomStat}, k, \text{Occ}) \\
 & & \Leftarrow x_1 \leq x \leq x_2 \wedge y_1 \leq y \leq y_2, \\
\ \ \text{RoomEmp} & = & (\text{RoomStat}, k, \mathit{status}) \mapsto (\text{RoomStat}, k, \text{Free}), \\
\ \ \text{TurnOff} & = & (\text{RoomStat}, k, \text{Free})? : (\text{Light}, k, \text{On}) \mapsto (\text{Light}, k, \text{Off}), \\
\ \ \text{TurnOn} & = & (\text{RoomStat}, k, \text{Occ})? : (\text{Light}, k, \text{Off}) \mapsto (\text{Light}, k, \text{On}) \\
\} & &
\end{array}
$$

In this program, rule RoomOcc changes the status of a room to occupied, if a person is detected in that room. Rule RoomEmp declares the room to be empty. (It is worth mentioning that *Gamma* rules cannot check for absence of an element, e.g., a person, in the multiset. This fact will be more clear when we give the semantics of *Gamma* programs in the next section.) Note that we have omitted the condition part of this rule. Throughout the rest of the paper, we leave out the condition part of a rule, if it is True. As their names suggest, rules TurnOff and TurnOn are in charge of turning the lights on and off. The functionality model of the system is defined as the union of the *Gamma* programs Movement and Control.

The above model specifies basic functionality units that different parts of the system can offer (e.g., sensor network, control subsystem and behavior of individuals). These form the basic ingredients of our system design. The semantics of *Gamma* programs allows for arbitrary executions of the above program called *the chaotic behavior*. The chaotic behavior contains all possible concurrent as well as sequential executions of rules. The chaotic behavior of the above functionality model has no assumption about timing of rules (e.g., the relationship between the speed of individuals and their position change is not specified). Furthermore, it abstracts from ordering and concurrency in execution of actions; that is, it contains several undesired patterns of behavior (e.g., turning the lights off with a person present in the room or putting extra limits on movements of individuals due to sequential execution of their movement rules). To specify these aspects of design and restrict the design model, we introduce timing and coordination modeling languages in the following sections.

3. Timing

3.1 Motivation

Abadi and Lamport [1994] argue that adding a new state variable named *now* will extend the semantics of un-timed systems to a real-time setting, in practice. However, the rest of their paper shows, ironically, that this new variable is too different from a normal state variable and that fixing the right semantics for it may need relatively long logical formulas (for example to express that time cannot decrease along a run).

In addition to the basic properties of time (as a state variable), the timed behavior of real-time systems is governed by their performance and timeliness specifications, which is a separate quality aspect. Thus, it makes sense to specify timing information (requirements) as a separate aspect and observe how it reflects on the semantics of the functional and coordination aspects when composed with their respective models.

This section provides a background on our timing aspect design. The timing aspect can itself be divided into a number of sub-aspects. A major classification in this area is to divide timing specifications into: performance- and deadline-related properties [Balir *et al.* 1998]. Performance-related properties are concerned with actual timing that processes have to spend on their resources (computation time, communication time, etc.). Deadline-related properties specify the desired real-time properties of the system and define its overall correctness criteria. In this paper, we mainly address performance-related properties and among them mainly computation time (nevertheless, the framework can be generalized to other performance properties as we point out in the remainder). However, we defer the specification of deadline-related properties to a temporal-logic-based specification language as we sketch in [Mousavi *et al.* 2002]. Enforcing these properties should be done through a refinement or synthesis technique. In the remainder, we first introduce preliminary notations and definitions needed to define timing. Then, we present our interval language and show how it can be combined with the functionality model by presenting an operational semantics for *Timed-Gamma*.

3.2 Basic definitions

Definition 6. (Basic time domain) *The* basic time domain *is a set denoted by* Time *on which a total ordering* $<$ *with least element* 0 *and an addition function* $+$ *with zero element* 0 *are defined.*

To model unspecified upper bounds of intervals, we introduce a new element to the basic time domain, denoted by ∞*, and refer to the new time domain as:* $\text{Time}_\infty \triangleq \text{Time} \cup \{\infty\}$. *We extend the ordering relation and addition function on* Time_∞ *such that for all* $t \in \text{Time}$, $t < \infty$, $t + \infty = \infty + t = \infty$, *and* $\infty + \infty = \infty$ *and not* $\infty < \infty$.

We do not need any more assumptions on the time domain Time. *Thus, we leave other properties of the time domain open (e.g., discreteness vs. denseness).*

An interval is a convex subset of Time_∞. We refer to the lower and upper bound of an interval I with $lb(I)$, and $ub(I)$, respectively.

Definition 7. (In-bound operator) *We define that* $t \in \text{Time}$ *is* in-bound *with respect to the interval* I*, denoted by* $t \prec I$ *if and only if* $t \leq lb(I) \vee t \in I$. *This operator is defined in order to avoid subtleties involved with having both open and close intervals.*

A timing specification is related to the functionality (and the coordination) domain by mentioning a rule name. The syntax of a timing specification is given in

Fig. 4. In this figure, the syntactic class `Time` represents the set Time. Note that the timing aspect is independent from the functionality aspects (and from the coordination aspect) and thus one may specify a functionality without specifying its timing and the other way round (a timing without specifying functionality). Here, for simplicity, we assumed that bounds of intervals are concrete numbers taken from the time domain. However, for more involved cases, timing may depend on other aspects of design and thus the intervals can be parameterized using information from other aspects of designs. In [Mousavi *et al.* 2002], we sketched the interaction between timing and distribution aspects (i.e., physical distribution of basic data items and their local stores, which can influence latencies in fetching data items).

```
TimingSpec  ::=  {Timings}
Timings     ::=  Timing | Timing, Timings
Timing      ::=  T_RuleName = Interval
Interval    ::=  [Time,Time] | (Time,Time] | [Time,Time)
              |  (Time,Time) | [Time,∞) | (Time,∞)
```

Fig. 4: Syntax of timing specification.

If there is no timing estimation specified for a rule (as is the case for un-timed specifications), it is assumed to be $[0, \infty)$, that is, an arbitrary computation time is indicated.

Example 5. (Light Control: Timing) Consider the functionality model of Example 4. We continue with the light control case study by specifying the timing aspect. Here, we assume that making a step to move within the building takes one unit of time (thus making a linear correlation between distance and speed). We assume that sensors can guarantee detecting the presence of a person in a room in between one to two units of time (depending on other applications using the sensor system). However, declaring a room to be empty is an internal action performed by the controller software (as a result of not receiving any presence report) and thus takes no time. All reporting actions are assumed to take some time between zero and one units. For the other rules, we leave the timing unspecified, either because we have no control on their timing behavior (since they are events caused by the users or the environment) or their timing is not crucial for the correctness of our system design.

$$\begin{array}{lll} T_{\mathrm{Move}} = [1,1] & T_{\mathrm{RoomOcc}} = [1,2] & T_{\mathrm{TurnOn}} = [0,1] \\ T_{\mathrm{Move}_d} = [1,1] & T_{\mathrm{RoomEmp}} = [0,0] & T_{\mathrm{TurnOff}} = [0,1] \end{array}$$

3.3 Semantics of timed functionality

In this subsection, we present the semantics of the composition of the aspects timing and functionality. The (syntactic as well as semantic) model resulting from this composition is called *Timed-Gamma*. The semantics of *Timed-Gamma* is presented in two parts. The first part, presented in Fig. 5, shows the basic timed-computation and termination of a rule. To maintain obliviousness of aspects, the semantics of unspecified rules is also specified in Fig. 5. Based on the semantics for rules, the rules in Fig. 6 define the behavior of *Timed-Gamma* programs. This part combines in a chaotic manner the behavior of the rules. The chaotic and abstract behavior of *Timed-Gamma* programs is defined in such a way that it leaves all implementation decisions open (e.g., scheduling and control policies, available resources and true level of concurrency).

3.3.1 Semantics of Gamma rules

The basic semantics of rules is defined in Fig. 5. In this semantics, it is assumed that there is a given *Timed-Gamma* program (R, I) where R is the *Gamma* program and I is the timing specification. The predicate func(rn) indicates whether or not a functionality is given for a rule with rule name rn, i.e., whether there is a definition for rule name rn in R. Also, $rn.R$ and $rn.I$ indicate the *Gamma* rule (if any) and timing specified for rule name rn in the *Gamma* program R and the timing specification I, respectively. If there is no interval defined for rn, $rn.I$ results in $[0, \infty)$. This gives us a timed semantics for *Gamma* programs in which no assumptions have been made about the timing of the functionality or in which no constraints exist for the timing of that functionality. For the time being, we assume that $rn.I$ works as a function. Nevertheless, this assumption could be relaxed by allowing several intervals associated to a rule, and hence $rn.I$ returning a set of time points which is not necessarily a single convex interval. This relaxed assumption would not require major change to our semantics. However, for simplicity we assume the single interval time paradigm from now on.

In the semantics two types of states are used. The state $\langle rn, M\rangle$ consists of the name of a *Gamma* rule rn and a multiset M (of shared data). The state $\langle rn[\alpha@t : I], M\rangle$ consists of an annotated *Gamma* rule, i.e., a *Gamma* rule rn annotated with a task $\alpha@t : I$, and a multiset M of shared data. A task $\alpha@t : I$ is the substitution α together with the elapsed computation time t (the duration that the task has been active and running till now), and the estimated computation time interval I. Note that a rule specifies a generic scheme for computation, while a task is a substitution, i.e., an instance of computation, that has been scheduled but not yet committed (possibly has not received enough processor time yet).

Predicate $\surd_{\mathrm{r}}$ stands for termination of a rule. The transitions in the given semantics are either of the form $\rightarrowtail_{\mathrm{r}}$ and $\overset{t}{\rightarrow}_{\mathrm{r}}$ (where $t > 0$ a time step which can range over the basic time domain Time) or $\overset{[\alpha]}{\rightarrow}_{\mathrm{r}}$, where α ranges over substitutions. A basic timed-computation is divided into three phases: scheduling, computation and commitment.

The first phase consists of scheduling a task (see deduction rules *(RuleSched0)* and *(RuleSched1)*). Deduction rule *(RuleSched0)* concerns scheduling a task for a defined *Gamma* rule in R. Deduction rule *(RuleSched1)* allows for scheduling an arbitrary task in case the definition of rule rn is not given by the *Gamma* program

$$\textit{(RuleSched0)}\frac{\text{func}(rn) \quad M, v \propto rn.R}{\langle rn, M\rangle \rightarrowtail_{\text{r}} \langle rn[\alpha @0 : rn.I], M\rangle}$$
where $rn.R$ is the form $rn = lhs \mapsto rhs \Leftarrow c$ and $\alpha = \overline{v}(rhs)/\overline{v}(lhs)$

$$\textit{(RuleSched1)}\frac{\neg\text{func}(rn) \quad M \models [\alpha]}{\langle rn, M\rangle \rightarrowtail_{\text{r}} \langle rn[\alpha @0 : rn.I], M\rangle}$$

$$\textit{(TimePass)}\frac{t + t' \prec I}{\langle rn[\alpha @t : I], M\rangle \overset{t'}{\longrightarrow}_{\text{r}} \langle rn[\alpha @t + t' : I], M\rangle}$$

$$\textit{(RuleComp)}\frac{t \in I}{\langle rn[\alpha @t : I], M\rangle \overset{[\alpha]}{\longrightarrow}_{\text{r}} \langle rn, M([\alpha])\rangle}$$

$$\textit{(RuleTerm0)}\frac{\text{func}(rn) \quad \neg\exists_v\, M, v \propto rn.R}{\langle rn, M\rangle\surd_{\text{r}}}$$

$$\textit{(RuleTerm1)}\frac{\neg\text{func}(rn)}{\langle rn, M\rangle\surd_{\text{r}}}$$

Fig. 5: Basic timed functionality: semantics of *Gamma* rules.

R. When a rule (with name) rn is defined in R, scheduling a task using this rule is only possible if there exists a valuation v that both satisfies the condition and valuates the left-hand-side expression of r to be a part of the multiset (denoted by $M, v \propto rn.R$).

Definition 8. (Enabling valuation) *A valuation v on variables enables a rule r of the form $rn = lhs \mapsto rhs \Leftarrow c$ for a multiset M, notation $M, v \propto r$, if and only if $\overline{v}(c)$ holds and $\overline{v}(lhs) \sqsubseteq M$. Here $\overline{v}$ denotes a valuation on expressions that is induced by the valuation v on variables.*

The scheduling of a task is indicated by a transition $\rightarrowtail_{\text{r}}$. We abstract from the time needed for finding an appropriate valuation. Alternatively, this time could be added to the transition associated with scheduling a task.

The second phase of basic computation is spending (processor) time on a computation, using rule *(TimePass)*. Note that the concept of timing is the same between specified and unspecified rules and thus this rule is shared between the first two parts of the semantics.

Finally, the third phase is committing a computation via *(RuleComp)*, which results in application of the substitution from the task that is committed to the multiset.

The division of a basic computation in three phases provides the possibility to put further details in each of these phases (e.g. specifying scheduling policy, providing timing information for distributed scheduling or commitment).

Rules *(RuleTerm0)* and *(RuleTerm1)* represent the possibility of termination of a rule when it cannot schedule any new task and does not have any active task to perform.

The semantics of *Gamma* rules is the smallest set of closed transitions that are provable from the above set of rules. For a formal definition of the transitions defined by a deduction system we refer to [Aceto *et al.* 2001;Mousavi *et al.* 2007].

3.3.2 *Chaotic semantics of* Gamma *programs*

The semantics, given in Fig. 6, specifies the general chaotic behavior of *Timed-Gamma* programs by composing behavior of rules from a certain set *RN* in all possible orders and all possible levels of true concurrency. This reflects the pure timed-functionality model of our system.

$$\textit{(ProgSched)}\frac{\langle rn, M\rangle \rightarrowtail_{\mathrm{r}} \langle rn[\alpha@0 : I], M\rangle \quad M \models [\alpha] \bowtie comp(T) \quad rn \in RN}{\langle RN, M, T\rangle \rightarrowtail_{\mathrm{p}} \langle RN, M, [\alpha@0 : I] \boxplus T\rangle}$$

$$\textit{(ProgTime0)}\frac{\langle rn[\alpha@t : I], M\rangle \xrightarrow{t'}_{\mathrm{r}} \langle rn[\alpha@t + t' : I], M\rangle \quad rn \in RN}{\langle RN, M, [\alpha@t : I] \boxplus T\rangle \xrightarrow{t'}_{\mathrm{p}} \langle RN, M, [\alpha@t + t' : I] \boxplus T\rangle}$$

$$\textit{(ProgTime1)}\frac{\langle RN, M, T_0\rangle \xrightarrow{t}_{\mathrm{p}} \langle RN, M, T_0'\rangle \quad \langle RN, M, T_1\rangle \xrightarrow{t}_{\mathrm{p}} \langle RN, M, T_1'\rangle}{\langle RN, M, T_0 \boxplus T_1\rangle \xrightarrow{t}_{\mathrm{p}} \langle RN, M, T_0' \boxplus T_1'\rangle}$$

$$\textit{(ProgComp0)}\frac{\langle rn[\alpha@t : I], M\rangle \xrightarrow{[\alpha]}_{\mathrm{r}} \langle rn, M'\rangle \quad rn \in RN}{\langle RN, M, [\alpha@t : I] \boxplus T\rangle \xrightarrow{[\alpha]}_{\mathrm{p}} \langle RN, M', T\rangle}$$

$$\textit{(ProgComp1)}\frac{\langle RN, M, T_0\rangle \xrightarrow{\sigma_0}_{\mathrm{p}} \langle RN, M_0', T_0'\rangle \quad \langle RN, M, T_1\rangle \xrightarrow{\sigma_1}_{\mathrm{p}} \langle RN, M_1', T_1'\rangle}{\langle RN, M, T_0 \boxplus T_1\rangle \xrightarrow{\sigma_0 \boxplus \sigma_1}_{\mathrm{p}} \langle RN, M(\sigma_0 \boxplus \sigma_1), T_0' \boxplus T_1'\rangle}$$

$$\textit{(ProgTerm)}\frac{\forall_{rn \in RN}\ \langle rn, M\rangle \surd_{\mathrm{r}}}{\langle RN, M, \varnothing\rangle \surd_{\mathrm{p}}}$$

Fig. 6: Timed functionality: semantics of *Timed-Gamma* programs.

In the semantics presented in Fig. 6, the states are of the form $\langle RN, M, T\rangle$ where *RN* is a set of rule names for which there may be functionality and timing specifications in R and I respectively, M is a multiset of shared data and T is a multiset of tasks. We use $\surd_{\mathrm{p}}$, $\rightarrowtail_{\mathrm{p}}$, $\xrightarrow{t}_{\mathrm{p}}$ and $\xrightarrow{\sigma}_{\mathrm{p}}$ instead of $\surd_{\mathrm{r}}$, $\rightarrowtail_{\mathrm{r}}$, $\xrightarrow{t}_{\mathrm{r}}$ and $\xrightarrow{[\alpha]}_{\mathrm{r}}$, respectively. The rule *(ProgSched)* shows that a program can schedule a new task if it can be scheduled by a rule and the task is independent from the current context of parallel tasks. The computations introduced in the task multiset are checked for consistency with respect to the data multiset (by means of an independence check with existing tasks). Here $comp(T)$ denotes the computation that is obtained by stripping from the tasks in the task multiset T all timing information. This way consistency of the task multiset is maintained during the execution of a *Gamma* program.

Rules *(ProgTime0)* and *(ProgTime1)* specify spending time on single and concurrent computations respectively. Rules *(ProgComp0)* and *(ProgComp1)* specify how a *Timed-Gamma* program can perform computations. The above four rules provide an abstraction from the true level of concurrency.

Rule *(ProgTerm)* decides on the termination of a program based on the termination of all constituting rules. It is worth mentioning that rule termination is not necessarily permanent; a terminated rule may become enabled later due to activity of other rules. Only if all rules have a defined functionality and they all terminate, termination becomes permanent.

The semantics of a *Timed-Gamma* program is the smallest transition relation satisfying the above transition rules. Note that the absence of an independence check in the premise of rules such as *(ProgComp1)* cannot introduce inconsistency because states with an inconsistent task set are not reachable (since *(ProgSched)* checks consistency in the introduction of new tasks and all other rules preserve the consistency of task sets).

We separated the two parts of the semantics in order to re-use the first part in both defining the chaotic behavior of *Timed-Gamma* programs (the second part) and also the coordinated behavior of schedules (in the next section). In other words, the first part of the semantics serves to define the basic units of functionality. Technically speaking, one can replace this particular model of timed functionality with an operational semantics of another functionality model (say, JavaSpace [Freeman *et al.* 1999] methods, or even a hierarchy of interface services), and benefit from the specification model presented in the remainder. In such cases, care should be taken in order not to loose the orthogonality in the model as it is presented here.

Example 6. (Scheduling and Executing Tasks) Consider the functionality and timing models specified in Examples 4 and 5. Assume that we start from multiset M containing two rooms with boundaries $(0, 0, 10, 10)$ and $(0, 10, 10, 20)$ (for both of which the lights are off), a door located at point (0,0), and only one person in position $(5, 5)$ which has been detected already to occupy room 1:

$$M = \begin{bmatrix} (\mathrm{RoomDim}, 1, 0, 0, 10, 10), (\mathrm{RoomStat}, 1, \mathrm{Occ}), (\mathrm{Light}, 1, \mathrm{Off}), \\ (\mathrm{RoomDim}, 2, 0, 10, 10, 20), (\mathrm{RoomStat}, 2, \mathrm{Free}), (\mathrm{Light}, 2, \mathrm{Off}), \\ (\mathrm{Door}, 5, 0), (\mathrm{Door}, 5, 10), \\ (\mathrm{Person}, 1, 5, 5) \end{bmatrix}.$$

The constant Speed is 5. The following sequence of transitions is a possible run of the timed-program resulting from composing the timing aspect with the programs Movement and Control. Let *RN* denote the set of all rule names that occur in those *Timed-Gamma* programs.

$$\begin{aligned}&\langle RN, M, \varnothing\rangle \rightarrowtail_{\mathrm{p}} \langle RN, M, [\alpha_0@0 : [1,1]]\rangle \overset{0.5}{\rightarrow}_{\mathrm{p}} \langle M, [\alpha_0@0.5 : [1,1]]\rangle \rightarrowtail_{\mathrm{p}}\\ &\langle M, [\alpha_0@0.5 : [1,1], \alpha_1@0 : [0,1]]\rangle \overset{0.5}{\rightarrow}_{\mathrm{p}} \langle M, [\alpha_0@1 : [1,1], \alpha_1@0.5 : [0,1]]\rangle \overset{0.5}{\rightarrow}_{\mathrm{p}}\\ &\langle M, [\alpha_0@1 : [1,1], \alpha_1@1 : [0,1]]\rangle \overset{[\alpha_0]}{\rightarrow}_{\mathrm{p}} \langle M([\alpha_0]), [\alpha_1@1 : [0,1]]\rangle \overset{[\alpha_1]}{\rightarrow}_{\mathrm{p}}\\ &\langle M([\alpha_0, \alpha_1]), \varnothing\rangle,\end{aligned}$$

where

$$\alpha_0 = [(\mathrm{Person}, 1, 7, 7)]/[(\mathrm{Person}, 1, 5, 5)]$$

and

$$\alpha_1 = [(\mathrm{RoomStat}, 1, \mathrm{Occ}), (\mathrm{Light}, 1, \mathrm{On})]/[(\mathrm{RoomStat}, 1, \mathrm{Occ}), (\mathrm{Light}, 1, \mathrm{Off})].$$

Note that in the above run the timing of rule Move is interleaved with the timing of rule TurnOn which is due to the general chaotic execution of *Timed-Gamma* programs. However, it is desirable to specify that the execution of the two rules is independent from each other (from the timing perspective, i.e., people can move while the lights are turning on). Thus, while the functional specification together with the timing aspect has an executable semantics, it fails to provide some essential liveness requirements of our system (though, it preserves some of the safety requirements such as that the light is not turned off for a room which has been identified to be occupied). To enforce these liveness properties, we have to devise a more precise execution strategy of *Gamma* rules which is left to the coordination aspect. This is the topic of our next section.

One can get a pure timing and a pure functionality model from the above semantics by leaving out details about computations (considering all computations unspecified) and data multiset or timing and task multiset, respectively. This can be done on the syntactical as well as the semantical level, however, we dispense with presenting the projected semantics here. Elsewhere, we have defined a pure functional semantics for *Gamma* programs [Mousavi *et al.* 2002].

4. Coordination

The aim of our coordination language, named *Schedules*, is to define the correct ordering, synchronization, and interaction of basic (timed) functionalities. Due to our design philosophy of separation of concerns, the aspect of coordination should be kept orthogonal with respect to timing to the largest possible extent. Thus, we assume a timed functionality model and use it as the basis of our coordination modeling language. A *schedule*, i.e., an expression in our coordination language, does not add information about timing and functionality. Furthermore it does not assume anything about specifications in other domains. If such specifications are present, their respective semantics are used to define the results of scheduling strategies. If specifications in other domains are absent a default timed-functionality is used instead. Thus, the underlying modeling languages as well as the underlying models can be changed independently (as long as they maintain the same structure on the transition system) and the result of their change will be reflected in the semantics of coordination indirectly.

4.1 Syntax of Schedules

The syntax of our language is specified in Fig. 7.

$$\begin{array}{lcl} \texttt{Schedule} & ::= & \texttt{RuleName} \mid \texttt{Schedule} \,;\, \texttt{Schedule} \\ & \mid & \texttt{Schedule} + \texttt{Schedule} \\ & \mid & \texttt{RuleName} \curvearrowright \texttt{Schedule[Schedule]} \\ & \mid & \texttt{Schedule} \parallel \texttt{Schedule} \mid \texttt{Schedule} \,|||\, \texttt{Schedule} \\ & \mid & \mu\texttt{RecursionVar}.\, \texttt{Schedule} \mid \texttt{RecursionVar} \end{array}$$

Fig. 7: Syntax of *Schedules*.

A rule from a *Timed-Gamma* program can be used as a building block for a schedule via its name. Sequential composition of schedules is denoted by ;. Nondeterministic choice between two schedules is specified using the + operator. The rule-conditional operator $\curvearrowright$ is used to select from different strategies based on whether or not a rule can be scheduled. Namely, in schedule $r \curvearrowright s_0[s_1]$ if rule r can be scheduled with respect to the current state of the multiset, then schedule s_0 is chosen for execution, otherwise, s_1 is executed. Abstract parallel composition ($\|$) allows for both concurrent and serialized execution of components (to represent the cases where there may or may not be enough resources for true concurrency). Strict parallel composition ($|||$) forces the participating components to run concurrently, i.e., to synchronize in their timing steps. Abstract parallel composition is suitable for cases where no information about available and needed resources is assumed and thus concurrent and serialized executions are both possible (e.g., high-level specifications, which should allow for different resource-allocation schemes). However, strict parallel composition is used for cases where the two components have a simultaneous and independent execution and should not be serialized (for example, to separate timing of the environment from the system, or concurrent tasks running on different processors). The recursion operator μ is used to make recursive schedules ($\mu x.s(x)$) explicitly using recursion variables (denoted by the syntactic class `RecursionVar`). Recursion can be used to build schedules which can repeat a certain behavior and exhibit infinite behavior. Only schedules in which all recursion variables are bound by μ are of interest in this paper. In the rest of this paper, we usually define a name for a schedule. Such a name serves as a syntactic shorthand for the defined schedule.

To make the basic *Gamma* theory more usable, we add some syntactic sugar for strengthening rule conditions. For each rule with name *rn* and for each condition *c*, we assume the existence of a rule with name $c \rhd rn$ as defined below.

DEFINITION 9. (STRENGTHENING CONDITION) *The strengthening of a rule* $rn = lhs \mapsto rhs \Leftarrow c_1$ *by a condition* c_2 *is defined to be the rule* $c_2 \rhd rn = lhs \mapsto rhs \Leftarrow c_1 \wedge c_2$. *Unless specified otherwise, the timing of the rule with name* $c_2 \rhd rn$ *is the timing of the rule with name rn.*

Next, we extend the notion of strengthening the condition of a rule from Definition 9 to schedules. The idea is that the strengthening condition *c* distributes to all rule names that occur in the schedule. This can be used for indicating that a certain rule may only be used for a certain instance of one of the variables that occur in that rule, e.g., $i = 1 \rhd$ Move indicates a rule that only describes the movement of the person with identity 1.

DEFINITION 10. (STRENGTHENING CONDITIONS: EXTENDED) *For an arbitrary schedule s, strengthening it with condition c, denoted by* $c \rhd s$ *is defined inductively as follows:*

(1) $c \rhd rn$, *if s is of the form rn, where rn is a* Timed-Gamma *rule name; Strengthening the condition of a rule is defined in Definition 9;*

(2) $c \rhd (s' \; op \; t')$, *for all schedules* s' *and* t' *and all binary operators op in the syntax of* Schedules *is defined as* $(c \rhd s') \; op \; (c \rhd t')$*;*

(3) $c \rhd (rn \curvearrowright s'[t'])$ *is defined as* $(c \rhd rn) \curvearrowright (c \rhd s')[c \rhd t']$, *for all rule names rn and schedules* s' *and* t'*;*

(4) $c \rhd (\mu x.s')$ *is defined as* $\mu x.(c \rhd s')$, *for all recursive variables x and schedules* s'*;*

(5) $c \rhd x$ *for all recursion variables x is defined as x.*

In the following example, we illustrate the syntax of *Schedules* by giving a schedule for the movement of individuals within the building.

EXAMPLE 7. The following schedule defines the movement behavior of individuals in the light-control case study. Individuals can move independently from each other. Movements of individuals can be done arbitrary order:

$$\text{IndMove} \;=\; |||_{1 \leq pid \leq \text{maxP}} \; ((i = pid) \rhd (\mu x. \; (\text{idle} \parallel (\text{Move} \parallel (\text{MoveIn} \parallel \text{MoveOut}))) \; ; x)).$$

Note that in this specification, maxP represents the number of persons and it is assumed that their identities range from 1 to maxP. The notation $|||_{1 \leq pid \leq \text{maxP}}$ is a shorthand for a number of composed parallel terms (assuming commutativity, associativity and skip (introduced below) as the identity element of strict parallel composition), namely one copy of the schedule $(i = pid) \rhd (\mu x.(\text{idle} \parallel (\text{Move} \parallel (\text{MoveIn} \parallel \text{MoveOut}))) \; ; x)$ for each $1 \leq pid \leq \text{maxP}$.

We are aware that our treatment of variables in the *Gamma* rules is rather primitive and a rigorous treatment of formal parameters for schedules and *Gamma* rules will make the programs more readable. However, we do not introduce this rather standard extension for sake of simplicity in the formal development of the paper.

4.2 Semantics of Schedules

Fig. 8 shows the first set of semantic rules for the timed-coordination language. In these deduction rules, *rn* and *arn* are (meta-)variables ranging over rule names and annotated rule names, respectively. As in the semantics of *Timed-Gamma*, these rules link the semantics of single-rule execution to the semantics of schedules (coordination terms). However, in the coordination semantics, there is a tight relationship between coordination terms and scheduled tasks. For example, to check synchronization requirements of tasks with similar substitutions with respect to parallel and sequential compositions (in a schedule like $s \parallel (s \; ; \; t)$ where the task corresponding to the second instance of s is a prerequisite for the task instantiated from t whilst the task corresponding to the first instance of s has no effect on other tasks). Hence, we also attach scheduled tasks of each coordination term to its respective syntactic expression (see rule *(CoordSched)*)[1]. Also observe that, in rule *(CoordComp)*, a rule, after committing a computation, is replaced by the rule skip, where skip will be defined shortly as the schedule that cannot schedule a new task.

$$\textit{(CoordSched)}\frac{\langle rn, M\rangle \rightarrowtail_{\mathrm{r}} \langle arn, M'\rangle}{\langle rn, M\rangle \rightarrowtail \langle arn, M'\rangle}$$

$$\textit{(CoordTimePass)}\frac{\langle arn, M\rangle \xrightarrow{t}_{\mathrm{r}} \langle arn', M'\rangle}{\langle arn, M\rangle \xrightarrow{t} \langle arn', M'\rangle}$$

$$\textit{(CoordComp)}\frac{\langle arn, M\rangle \xrightarrow{\sigma}_{\mathrm{r}} \langle rn, M'\rangle}{\langle arn, M\rangle \xrightarrow{\sigma} \langle \mathrm{skip}, M'\rangle}$$

$$\textit{(CoordRuleTerm)}\frac{\langle rn, M\rangle\checkmark_{\mathrm{r}}}{\langle rn, M\rangle\checkmark}$$

Fig. 8: Semantics of timed coordination: basic computation.

So, in the given semantics, the state $\langle s, M\rangle$ contains s as the coordination expression that is possibly augmented with scheduled tasks (substitution, timing, and interval) and M is the data multiset, as before. Similar predicates and transition relations are used as in the previous semantics.

[1] From a process algebraic point of view this means that we introduce $r[\alpha@t : I]$ as a new schedule term for rule name r, substitution α, time point t and interval I.

Next, the semantics of the schedule composition operators are defined one by one using deduction rules. In the deduction rules, a transition $\overset{\chi}{\rightsquigarrow}$ either denotes scheduling a task, passage of time or performing a computation (χ is a variable that ranges over the (basic) time domain and computations).

Skip and idle

The schedule skip is the rule name for a *Timed-Gamma* rule that is never enabled and the schedule idle is the rule name for a rule that is always enabled but makes no change in the multiset, i.e., it has empty expressions on both sides. The timing specification of skip is not of any interest, for the timing of idle it is assumed that there are no restrictions, thus the interval $[0, \infty)$ is assumed. The semantics of these schedules is obtained from the following identification with *Gamma*-rules and Timing specifications with the same name.

DEFINITION 11. (SKIP AND IDLE) *The rules* skip *and* idle *are defined as follows:*

- ∘ $\text{skip} = \epsilon \mapsto \epsilon \Leftarrow \text{False}$ *and* $T_{\text{skip}} = [0, \infty)$*;*
- ∘ $\text{idle} = \epsilon \mapsto \epsilon \Leftarrow \text{True}$ *and* $T_{\text{idle}} = [0, \infty)$.

Rule conditional

Rules *(RC0)* to *(RC3)* define the semantics for the rule-conditional operator. Expression $rn \curvearrowright s[t]$ can schedule a task when either the rule with name rn is enabled and the first argument s can schedule a task, or when the rule with name rn terminates (for rules with functionality this is equivalent to saying that the rule is disabled at the moment) and t can schedule a task. Obviously, it terminates when none of the above cases are possible.

$$\textit{(RC0)}\frac{\langle rn, M\rangle \rightarrowtail_{\mathrm{r}} \quad \langle s, M\rangle \rightarrowtail \langle s', M'\rangle}{\langle rn \curvearrowright s[t], M\rangle \rightarrowtail \langle s', M'\rangle}$$

$$\textit{(RC1)}\frac{\langle rn, M\rangle\surd_{\mathrm{r}} \quad \langle t, M\rangle \rightarrowtail \langle t', M'\rangle}{\langle rn \curvearrowright s[t], M\rangle \rightarrowtail \langle t', M'\rangle}$$

$$\textit{(RC2)}\frac{\langle rn, M\rangle \rightarrowtail_{\mathrm{r}} \quad \langle s, M\rangle\surd}{\langle rn \curvearrowright s[t], M\rangle\surd} \qquad \textit{(RC3)}\frac{\langle rn, M\rangle\surd_{\mathrm{r}} \quad \langle t, M\rangle\surd}{\langle rn \curvearrowright s[t], M\rangle\surd}$$

In the above deduction rules the notation $\langle rn, M\rangle \rightarrowtail_{\mathrm{r}}$ represents $\langle rn, M\rangle \rightarrowtail_{\mathrm{r}} \langle arn, M'\rangle$ for some arn and M'.

Usually, checking for schedulability of a rule proceeds with executing the rule. Thus, we define the pattern $rn \rightarrow s[t]$ as a shorthand for $rn \curvearrowright (rn \,;\, s)[t]$.

Sequential composition

The semantics of sequential composition is specified by the following three rules.

$$\textit{(S0)}\frac{\langle s_0, M\rangle \overset{X}{\rightsquigarrow} \langle s_0', M'\rangle}{\langle s_0 \; ; \; s_1, M\rangle \overset{X}{\rightsquigarrow} \langle s_0' \; ; \; s_1, M'\rangle}$$

$$\textit{(S1)}\frac{\langle s_0, M\rangle\checkmark \quad \langle s_1, M\rangle \overset{X}{\rightsquigarrow} \langle s_1', M'\rangle}{\langle s_0 \; ; \; s_1, M\rangle \overset{X}{\rightsquigarrow} \langle s_1', M'\rangle} \qquad \textit{(S2)}\frac{\langle s_0, M\rangle\checkmark \quad \langle s_1, M\rangle\checkmark}{\langle s_0 \; ; \; s_1, M\rangle\checkmark}$$

Nondeterministic choice

Scheduling a task from a schedule resolves the nondeterministic choice among the two schedules.

$$\textit{(C0)}\frac{\langle s_0, M\rangle \rightarrowtail \langle s_0', M'\rangle}{\begin{array}{c}\langle s_0 + s_1, M\rangle \rightarrowtail \langle s_0', M'\rangle \\ \langle s_1 + s_0, M\rangle \rightarrowtail \langle s_0', M'\rangle\end{array}} \qquad \textit{(C1)}\frac{\langle s_0, M\rangle\checkmark}{\begin{array}{c}\langle s_0 + s_1, M\rangle\checkmark \\ \langle s_1 + s_0, M\rangle\checkmark\end{array}}$$

Abstract parallel composition

Rules *(P0)* to *(P4)* specify the semantics of the abstract parallel composition operator. This type of parallel composition does not enforce concurrent execution of tasks and allows them to be performed sequentially. In particular, rules *(P0)* and *(P1)* specify how two sides of a parallel composition can evolve individually. The side condition of *(P0)* assures that the task multiset remains consistent if either of the two sides schedules a new task. The computation that has been scheduled is obtained from an annotated coordination expression by the function *comp*.

DEFINITION 12. (COMPUTATION) *The computation of an annotated schedule, notation comp is defined as follows:*

$$\begin{array}{lcl}
comp(rn) & = & \varnothing \\
comp(rn[\alpha @ t : I]) & = & [\alpha] \\
comp(s \; ; \; s') & = & comp(s) \\
comp(s + s') & = & comp(s) \boxplus comp(s') \\
comp(rn \curvearrowright s[s']) & = & \varnothing \\
comp(s \parallel s') & = & comp(s) \boxplus comp(s') \\
comp(s \parallel\!| \; s') & = & comp(s) \boxplus comp(s') \\
comp(\mu x.s) & = & comp(s) \\
comp(x) & = & \varnothing .
\end{array}$$

(P2) specifies concurrent execution of the two sides by allowing them to spend time synchronously. Rule *(P3)* represents concurrent commitment of tasks. Rule *(P4)* specifies termination of abstract parallel composition.

$$\textit{(P0)}\frac{\langle s_0, M\rangle \rightarrowtail \langle s_0', M'\rangle \quad M' \models comp(s_0') \bowtie comp(s_1)}{\begin{array}{c}\langle s_0 \parallel s_1, M\rangle \rightarrowtail \langle s_0' \parallel s_1, M'\rangle \\ \langle s_1 \parallel s_0, M\rangle \rightarrowtail \langle s_1 \parallel s_0', M'\rangle\end{array}}$$

$$\textit{(P1)}\frac{\langle s_0, M\rangle \overset{\chi}{\rightsquigarrow} \langle s_0', M'\rangle \quad \chi \neq 0}{\begin{array}{c}\langle s_0 \parallel s_1, M\rangle \overset{\chi}{\rightsquigarrow} \langle s_0' \parallel s_1, M'\rangle \\ \langle s_1 \parallel s_0, M\rangle \overset{\chi}{\rightsquigarrow} \langle s_1 \parallel s_0', M'\rangle\end{array}}$$

$$\textit{(P2)}\frac{\langle s_0, M\rangle \xrightarrow{t} \langle s_0', M_0\rangle \quad \langle s_1, M\rangle \xrightarrow{t} \langle s_1', M_1\rangle}{\langle s_0 \parallel s_1, M\rangle \xrightarrow{t} \langle s_0' \parallel s_1', M\rangle}$$

$$\textit{(P3)}\frac{\langle s_0, M\rangle \xrightarrow{\sigma_0} \langle s_0', M_0\rangle \quad \langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle}{\langle s_0 \parallel s_1, M\rangle \xrightarrow{\sigma_0 \boxplus \sigma_1} \langle s_0' \parallel s_1', M(\sigma_0 \boxplus \sigma_1)\rangle}$$

$$\textit{(P4)}\frac{\langle s_0, M\rangle\checkmark \quad \langle s_1, M\rangle\checkmark}{\langle s_0 \parallel s_1, M\rangle\checkmark}$$

Strict parallel composition

Strict parallelism only differs from the abstract parallel composition operator in that it does not allow one component to prohibit or delay the other one in execution. In other words, it models true concurrency in which composed processes perform their behavior independent of each other under some global consistency conditions. To model this type of composition, the timing behavior of the parallel composition is restricted to allow time passage only when one of the parties cannot perform an action. Thus, tasks are forced to spend their computation time concurrently.

$$\textit{(SP0)}\frac{\langle s_0, M\rangle \rightarrowtail \langle s_0', M\rangle \quad M \models comp(s_0') \bowtie comp(s_1)}{\begin{array}{c}\langle s_0 \parallel\!| s_1, M\rangle \rightarrowtail \langle s_0' \parallel\!| s_1, M\rangle \\ \langle s_1 \parallel\!| s_0, M\rangle \rightarrowtail \langle s_1 \parallel\!| s_0', M\rangle\end{array}}$$

$$\textit{(SP1)}\frac{\langle s_0, M\rangle \xrightarrow{t} \langle s_0', M\rangle \quad \forall_{t'} \langle s_1, M\rangle \overset{t'}{\nrightarrow}}{\begin{array}{c}\langle s_0 \parallel\!| s_1, M\rangle \xrightarrow{t} \langle s_0' \parallel\!| s_1, M\rangle \\ \langle s_1 \parallel\!| s_0, M\rangle \xrightarrow{t} \langle s_1 \parallel\!| s_0', M\rangle\end{array}}$$

$$\textit{(SP2)}\frac{\langle s_0, M\rangle \xrightarrow{t} \langle s_0', M_0\rangle \quad \langle s_1, M\rangle \xrightarrow{t} \langle s_1', M_1\rangle}{\langle s_0 \parallel\!| s_1, M\rangle \xrightarrow{t} \langle s_0' \parallel\!| s_1', M\rangle}$$

$$\textit{(SP3)}\frac{\langle s_0, M\rangle \xrightarrow{\sigma_0} \langle s_0', M_0\rangle \quad \langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle}{\langle s_0 \parallel\!| s_1, M\rangle \xrightarrow{\sigma_0 \boxplus \sigma_1} \langle s_0' \parallel\!| s_1', M(\sigma_0 \boxplus \sigma_1)\rangle}$$

$$\textit{(SP4)}\frac{\langle s_0, M\rangle\checkmark \quad \langle s_1, M\rangle\checkmark}{\langle s_0 \parallel\!| s_1, M\rangle\checkmark}$$

Recursion

Finally, *(R0)* and *(R1)* specify the concept of recursion. Recursion is interpreted as replacing the recursion variable with the recursive term. Note that since recursion is not necessarily guarded in our language, it is possible to specify schedules that can neither make a transition, nor terminate (*deadlock* schedules such as $\mu x.x$).

$$\boxed{(R0)\frac{\langle s(\mu x.s/x), M\rangle \overset{X}{\rightsquigarrow} \langle s', M'\rangle}{\langle \mu x.s, M\rangle \overset{X}{\rightsquigarrow} \langle s', M'\rangle} \qquad (R1)\frac{\langle s(\mu x.s/x), M\rangle\checkmark}{\langle \mu x.s, M\rangle\checkmark}}$$

Note that $s(t/x)$ for recursion variable x and schedules s and t denotes the substitution of t for all free occurrences of x in s.

Example 8. We devise a control strategy for the timed-functionality model of Examples 4 and 5. The following schedule allows for a control strategy for managing lights and computing reports. The control loop consists of first declaring all rooms empty (initialization), then changing the status of rooms depending on the positions of persons in the building.

$$\begin{aligned}\text{Control} \;=\; & (\mu x.(\text{RoomEmp} \curvearrowright (\text{RoomEmp} \parallel x)))\;;\\ & (\mu y.(\parallel_{1\le rid\le \text{maxR}} ((k = rid) \rhd (\text{RoomOcc} \to \text{TurnOn}[\text{TurnOff}])))\;;\; y)\end{aligned}$$

where maxR denotes the maximum number of rooms.

This schedule for managing the lights is then composed with the schedule for the independent movement of individuals from Example 7 as follows:

$$\text{System} \;=\; \text{Movement} \;|||\; \text{Control}.$$

5. Notions of equality and refinement

In this section, we study initial notions of refinement and equality between shared-data-space applications (specified in terms of schedules or programs, possibly starting from an initial state). Different sorts of *simulation* and *bi-simulation* have been proposed as basic notions of refinement and equality for reactive systems to date [Park 1981; Milner 1980; Glabbeek 1993]. However, their extension to our setting in which the data part plays a role is not trivial and some design decisions have to be made in order to find the right notion of refinement and equality. An important goal driving our decision for a notion of equivalent and refinement is to have the *(pre-)congruence* property (so-called *robustness* with respect to interference from the environment), that is, we want algebraic equalities and refinement rules to hold in an arbitrary context. In this section, we introduce some possible notions of simulation and bisimulation (from Mousavi *et al.* [2004, 2005]) and study their properties (especially from the congruence point of view). To provide a general starting point for the refinement process, we define a *most general schedule* that induces the most general (chaotic) behavior of a functionality model and prove it to be the most general with respect to our strongest notion of refinement.

Our approach to refinement and equality is based on the well-known notions of simulation and bisimulation, respectively. We are aiming at using these notions in a setting where a complete specification of the environment (other components to be designed) is not available. As a first attempt, we define the following notion of statebased (bi-)simulation. Statebased simulation expresses that the refining schedule only performs transitions that the original schedule is able to perform and similarly, the refining schedule terminates only when the original schedule has an option of termination. Similarly statebased bisimulation specifies that both schedules always have the same options for computations and termination.

DEFINITION 13. (STATEBASED (BI-)SIMILARITY) *Relation R is called a statebased simulation relation on states (based on schedules) if and only if for all pairs $(\langle s, M\rangle, \langle t, M\rangle) \in R$:*

(1) $\forall_{X,s',M'} \langle s, M\rangle \overset{X}{\rightsquigarrow} \langle s', M'\rangle \Rightarrow \exists_{t'} \langle t, M\rangle \overset{X}{\rightsquigarrow} \langle t', M'\rangle$ and $(\langle s', M'\rangle, \langle t', M'\rangle) \in R$;

(2) $\langle s, M\rangle\checkmark \Rightarrow \langle t, M\rangle\checkmark$.

A symmetric statebased simulation relation is called a statebased bisimulation relation. Schedule s is called statebased similar[2] *to t with respect to initial state M, denoted by $s \leqq_M t$, if and only if there exists a statebased simulation relation R such that $(\langle s, M\rangle, (\langle t, M\rangle) \in R$. Statebased bisimilarity of s and t with respect to M is denoted by $s \underline{\leftrightarrow}_M t$ and is defined similarly.*

For most practical applications, the notion of statebased bisimulation is not a congruence since it relies on a particular initial state. The following example illustrates this fact.

EXAMPLE 9. Consider the two rules MoveOut and skip with respect to the multiset $[(\text{PersonOut}, 1), (\text{Door}, 10, 10)]$. It trivially holds that MoveOut $\underline{\leftrightarrow}_M$ skip. However it does not hold that MoveIn ; MoveOut $\underline{\leftrightarrow}_M$ MoveIn ; skip. Since the former does a step and transforms the multiset to $[(\text{Person}, 1, 10, 10), (\text{Door}, 10, 10)]$ and then the rule MoveOut becomes enabled and transforms the multiset to its original shape, while the latter can only perform the first transformation and terminates.

The following generalization of statebased (bi-)similarity detaches the notion from the initial multiset:

DEFINITION 14. (INITIALLY STATELESS (BI-)SIMILARITY) *Schedules s and t are called initially stateless (bi-)similar, notation $s \leqq_{\text{isl}} t$ ($s \underline{\leftrightarrow}_{\text{isl}} t$) if and only if there exists a statebased (bi-)simulation relation such that for all $M \in \mathbb{M}(U)$, $(\langle s, M\rangle, \langle t, M\rangle) \in R$.*

The above notion of bisimulation proves to be successful only for the sequential subset of the language *Schedules*:

[2] In the work of amongst others van Glabbeek [Glabbeek 2001], similarity is an equivalence. In our terminology, however, similarity is not an equivalence relation, but only a preorder.

THEOREM 1. *Initially stateless similarity is a pre-congruence and initially stateless bisimilarity is a congruence with respect to the sequential subset of* Schedules *(closed terms not containing parallel composition operators).*

PROOF. Our semantic rules fit the *sfisl* format of Mousavi *et al.* [2004, 2005], thus the congruence result for stateless bisimilarity follows from the corresponding theorem in Mousavi *et al.* [2004, 2005]. It is easy to see that this format also guarantees pre-congruence of initially stateless similarity. □

The above theorem can be useful in practice when we can decompose the schedule into sequential components and then use the initially stateless notions of bisimilarity and similarity for proving equivalence and refinement relations. However, when such a sequential decomposition is not possible the initially stateless notion of bisimulation fails to provide desired robustness property. The following example illustrates this fact.

EXAMPLE 10. Consider the following two schedules:

$$\begin{aligned} evac_1 &= (\mu x.\mathrm{MoveOut}_1 \ ; x) \ ; \mathrm{MoveOut}_1, \\ evac_2 &= (\mu x.\mathrm{MoveOut}_1 \ ; x) \ ; \mathrm{MoveOut}_2, \end{aligned}$$

where

$$\begin{aligned} &\mathrm{MoveOut}_1 = \mathrm{MoveOut}_2 = \mathrm{MoveOut}, \\ &T_{\mathrm{MoveOut}_1} = [1,2] \quad T_{\mathrm{MoveOut}_2} = [1,3]. \end{aligned}$$

Note that we have used two copies of the rule MoveOut because we have the desire to associate different timing specifications with different occurrences of this functionality.

Then these schedules $evac_1$ and $evac_2$ are initially stateless bisimilar:

$$evac_1 \underline{\leftrightarrow}_{\mathrm{isl}}\, evac_2.$$

The first recursive parts of $evac_1$ and $evac_2$ make sure that the building is evacuated from all persons thus the second parts always terminate immediately. However, composing these two schedules in abstract parallel composition with a single rule that allows for a person to enter the building reveals the difference between the two:

$$evac_1 \parallel \mathrm{MoveIn} \,\underline{\not\leftrightarrow}_{\mathrm{isl}}\, evac_2 \parallel \mathrm{MoveIn}.$$

This is due to the fact that in the execution of both schedules there is a path in which after evacuation of the building, a person enters it and then the first schedule, forces the person to move out within the interval $[1,2]$ while the latter allows the person to stay a bit longer. (It can perform MoveOut within the interval $(2,3]$.)

To make the (bi-)simulation robust in all possible environments, we introduce the following stateless (bi-)simulation.

DEFINITION 15. (STATELESS (BI-)SIMILARITY) *Relation R is called a stateless simulation relation on schedules if and only if for all pairs $(s,t) \in R$:*

(1) $\forall_{M,\mathcal{X},s',M'} \langle s, M\rangle \overset{\mathcal{X}}{\rightsquigarrow} \langle s', M'\rangle \Rightarrow \exists_{t'} \langle t, M\rangle \overset{\mathcal{X}}{\rightsquigarrow} \langle t', M'\rangle$ *and* $(s', t') \in R$*;*

(2) $\forall_M \langle s, M\rangle\checkmark \Rightarrow \langle t, M\rangle\checkmark$*.*

A symmetric stateless simulation relation is called a stateless bisimulation relation. We call two schedules s and t stateless (bi-)similar, notation $s \leqq t$ *(*$s \,\underline{\leftrightarrow}\, t$*), if and only if there exists a stateless (bi-)simulation relation R such that* $(s, t) \in R$*.*

Next, we state that strong bisimulation is indeed robust, with respect to all possible environments built upon the syntax of *Schedules*.

THEOREM 2. *Stateless (bi-)similarity is a pre-congruence (congruence) with respect to all operators in the syntax of* Schedules*.*

PROOF. Congruence of bisimilarity follows from the congruence result for the process-tyft format of [Mousavi *et al.* 2004]. It is easy to see that this format also guarantees pre-congruence of stateless similarity. □

The following proposition gives some algebraic rules that hold with respect to the notion of stateless (bi-)simulation.

PROPOSITION 1. *According to the given semantics, the following stateless similarities and bisimilarities hold, for all schedules* s, s_0, s_1, s_2 *and rule names rn:*

$$\begin{array}{rcl}
\text{skip} ; s & \underline{\leftrightarrow} & s \\
s ; \text{skip} & \underline{\leftrightarrow} & s \\
(s_0 ; s_1) ; s_2 & \underline{\leftrightarrow} & s_0 ; (s_1 ; s_2)
\end{array}
\qquad
\begin{array}{rcl}
s_0 + s_1 & \underline{\leftrightarrow} & s_1 + s_0 \\
(s_0 + s_1) + s_2 & \underline{\leftrightarrow} & s_0 + (s_1 + s_2)
\end{array}$$

$$\begin{array}{rcl}
\text{skip} \parallel s & \underline{\leftrightarrow} & s \\
s_0 \parallel s_1 & \underline{\leftrightarrow} & s_1 \parallel s_0 \\
(s_0 \parallel s_1) \parallel s_2 & \underline{\leftrightarrow} & s_0 \parallel (s_1 \parallel s_2)
\end{array}
\qquad
\begin{array}{rcl}
\text{skip} \mid\mid\mid s & \underline{\leftrightarrow} & s \\
s_0 \mid\mid\mid s_1 & \underline{\leftrightarrow} & s_1 \mid\mid\mid s_0 \\
(s_0 \mid\mid\mid s_1) \mid\mid\mid s_2 & \underline{\leftrightarrow} & s_0 \mid\mid\mid (s_1 \mid\mid\mid s_2)
\end{array}$$

$$\begin{array}{rcl}
rn \curvearrowright s[s] & \underline{\leftrightarrow} & s \\
rn \curvearrowright rn[\text{skip}] & \underline{\leftrightarrow} & rn \\
(rn \curvearrowright s_0[s_1]) ; s_2 & \underline{\leftrightarrow} & rn \curvearrowright (s_0 ; s_2)[s_1 ; s_2] \\
rn \curvearrowright (rn \curvearrowright s_0[s_1])[s_1] & \underline{\leftrightarrow} & rn \curvearrowright s_0[s_1]
\end{array}$$

$$\begin{array}{rcll}
\mu x.s & \underline{\leftrightarrow} & s(\mu x.s/x) & \\
\mu x.s & \underline{\leftrightarrow} & \mu y.s(y/x) & \textit{if } y \textit{ does not occur in } s
\end{array}$$

$$\begin{array}{rcl}
s_0 & \leqq & s_0 + s_1 \\
rn \curvearrowright s_0[s_1] & \leqq & s_0 + s_1 \\
s_0 \mid\mid\mid s_1 & \leqq & s_0 \parallel s_1
\end{array}$$

Proof. See Appendix A. □

The above congruence theorem has a very strong message for the provable equality and refinement relations, such as those proved in Proposition 1, namely, that these relations can be applied as equality and refinement rules in an arbitrary context. In other words, it allows for compositional reasoning and refinement of schedules. We use this fact in the following example to improve the control strategy proposed in the previous section.

Example 11. Recall the following schedule from Example 8:

$$\begin{aligned}\text{Control} = {} & (\mu x.(\text{RoomEmp} \curvearrowright (\text{RoomEmp} \parallel x)))\ ;\\ & (\mu y.(\|_{1\leq rid\leq \text{maxR}}\ ((k = rid) \triangleright (\text{RoomOcc} \to \text{TurnOn}[\text{TurnOff}])))\ ;\ y).\end{aligned}$$

One can prove that

$$\begin{aligned}\text{Control}' = {} & (\mu x.(\text{RoomEmp} \curvearrowright (\text{RoomEmp} \parallel x)))\ ;\\ & (\mu y.(\textstyle\prod_{1\leq rid\leq \text{maxR}}((k = rid) \triangleright (\text{RoomOcc} \to \text{TurnOn}[\text{TurnOff}])))\ ;\ y),\end{aligned}$$

where $\prod$ denotes a generalized sequential composition, is a refinement of schedule Control.

The refined schedule does not have a starvation problem because it orders the treatment of rooms by their numbers and serves them once in each control period. Thus, one can calculate the worst case response time for the control of each room using the upper bounds of the timing intervals. Thus, we can define the new system as:

$$\text{System}' = \text{Movement} \parallel\!| \text{Control}'.$$

Then, it follows from Theorem 2 that the following refinement relation holds:

$$\text{System}' \leqq \text{System}.$$

This means that the new system, in addition to providing new liveness (fairness) properties, preserves safety properties of the old system.

6. Chaotic behavior and the most general schedule

A general starting point for the refinement process of schedules is the chaotic behavior of *Gamma* programs which leaves all possible choices open. However, we do not have a representation of this chaotic behavior in the *Schedules* syntax. We aim at providing such a representation in the remainder.

Definition 16. (The Most General Schedule) *For a finite and non-empty set of Gamma rule names RN, we define its corresponding* most general schedule *as follows:*

$$\text{MGS}(RN) = \ \|_{rn\in RN}\ \mu x.rn \curvearrowright (rn \parallel x).$$

Define stateless bisimilarity between a *Gamma* program and a schedule in the obvious way. The following theorem expresses that the chaotic behaviour associated with a set of *Gamma* rules *RN* is captured precisely be the most general schedule MGS(*RN*).

THEOREM 3. *For any Gamma program RN, we have RN* $\underline{\leftrightarrow}$ MGS(*RN*).

PROOF. For the purpose of the sketch of the proof the equivalence of the Most General Schedule of a *Gamma* program and the *Gamma* program itself, we introduce the following notations:

(1) Assume that $RN = \{rn_1, \cdots, rn_n\}$ with all mentioned rules different.

(2) For some rule name rn and task multiset T, we use MGS(rn, T) to denote the set of all schedules of the form

$$rn[\alpha_1 @ t_1 : I_1] \parallel (\cdots (rn[\alpha_m @ t_m : I_m] \parallel \mu x.rn \curvearrowright (rn \parallel x)) \cdots)$$

where $comp(T) = [\alpha_1, \cdots, \alpha_m]$.

(3) For some schedule s and task multiset T, we use MGS(RN, T) to denote the set of all schedules of the form

$$\parallel_{1 \le i \le n} s_i$$

where $T = T_1 \boxplus \cdots \boxplus T_n$ and $s_i \in \text{MGS}(rn_i, T_i)$ for each $1 \le i \le n$.

The idea is to relate a program with task multiset T with each of the schedules from MGS(RN, T). This is then a stateless bisimulation relation up-to the following properties:

$$\begin{array}{lcl} \text{skip} \parallel s & \underline{\leftrightarrow} & s \\ s_0 \parallel s_1 & \underline{\leftrightarrow} & s_1 \parallel s_0 \\ (s_0 \parallel s_1) \parallel s_2 & \underline{\leftrightarrow} & s_0 \parallel (s_1 \parallel s_2) \end{array}$$

From this it follows that *RN* and MGS(*RN*) are stateless bisimilar. □

7. Conclusion and future work

In this paper, we presented a formal framework for separation of concerns in the design of real-time reactive systems. The basic aspects of design addressed in this paper are functionality, coordination and timing. For each aspect a language is formally defined such that semantics of the languages can be composed. For functionality, we used a shared-data-space paradigm to capture basic computational units (i.e., functionalities of components). For the coordination aspect, we used a process-algebraic formalism to specify the control skeletons. Finally, for timing aspects, we used intervals for specification of computation time. Each of these aspects can be extended or replaced by a different aspect language in its own right as long as the extended / replaced aspects can provide the expected transition system semantics. Further, we presented notions of equality and refinement for reasoning about systems developed in this framework. The notions of equality and refinement are built up in such a way that they can be deployed in an arbitrary environment and remain robust in spite of interferences from such an environment.

Our work forms the mathematical basis for a correct-by-construction method of system design. Such a method starts with a set of declarative specifications about aspects of design, massages each of these aspect specifications to a correct specification with respect to some overall correctness criterion, and finally transforms the specification to an executable code in an implementation framework. Such an implementation framework could be a middleware with support for different concerns, such as, for example, [Russello *et al.* 2004]. A concrete challenge in this direction is the synthesis of a correct-by-construction schedule given a set of (real-time) correctness criteria.

Another challenge is how to define languages tailored for different aspects such that specifications in these languages can be composed and refined independently.

Appendix A. Proof of proposition 1

In this appendix, AS denotes the set of all closed (annotated) schedule terms and Id denotes the relation $\{(s, s) \mid s \in AS\}$.

We only give the stateless bisimulation and simulation relations.

For the properties involving the schedule skip it is convenient to use the deduction rule

$$(Skip)\frac{}{\langle \text{skip}, M\rangle \checkmark}$$

instead of its definition as a special rule (see Definition 11). It can easily be shown that the schedule skip as defined by the above deduction rule is precisely the same as the one defined in Definition 11.

In the proof of skip $\| \; s \leftrightarroweq s$ we have used the property that

$$\langle s, M\rangle \rightarrowtail \langle s', M'\rangle \;\Rightarrow\; M' \models comp(s')$$

for any annotated schedules s and s', and any multisets M and M'. The property is easily proven by induction on the depth of the derivation of the antecedent.

skip ; $s \leftrightarroweq s$: The symmetric closure of the relation $R = \{(\text{skip} \; ; \; s, s)\} \cup Id$ is a stateless bisimulation relation.

s ; skip $\leftrightarroweq s$: The symmetric closure of the relation $R = \{(t \; ; \text{skip}, t) \mid t \in AS\}$ is a stateless bisimulation relation.

$(s_0 \; ; \; s_1) \; ; \; s_2 \leftrightarroweq s_0 \; ; \; (s_1 \; ; \; s_2)$: The symmetric closure of the relation $R = \{((t_0 \; ; \; s_1) \; ; \; s_2, t_0 \; ; \; (s_1 \; ; \; s_2)) \mid t_0 \in AS\} \cup Id$ is a stateless bisimulation relation.

$s_0 + s_1 \leftrightarroweq s_1 + s_0$: The relation $R = \{(s_0 + s_1, s_1 + s_0)\} \cup Id$ is a stateless bisimulation relation.

$(s_0 + s_1) + s_2 \leftrightarroweq s_0 + (s_1 + s_2)$: The symmetric closure of the relation $R = \{((s_0 + s_1) + s_2, s_0 + (s_1 + s_2))\} \cup Id$ is a stateless bisimulation relation.

$\text{skip} \parallel s \underline{\leftrightarrow} s$: The symmetric closure of the relation $R = \{(\text{skip} \parallel t, t) \mid t \in AS\}$ is a stateless bisimulation relation.

$s_0 \parallel s_1 \underline{\leftrightarrow} s_1 \parallel s_0$: The relation $R = \{(t_0 \parallel t_1, t_1 \parallel t_0) \mid t_0, t_1 \in AS\}$ is a stateless bisimulation relation.

$(s_0 \parallel s_1) \parallel s_2 \underline{\leftrightarrow} s_0 \parallel (s_1 \parallel s_2)$: The symmetric closure of the relation $R = \{((t_0 \parallel t_1) \parallel t_2, t_0 \parallel (t_1 \parallel t_2)) \mid t_0, t_1, t_2 \in AS\}$ is a stateless bisimulation relation.

$\text{skip} \parallel\!| s \underline{\leftrightarrow} s$: The symmetric closure of the relation $R = \{(\text{skip} \parallel\!| t, t) \mid t \in AS\}$ is a stateless bisimulation relation.

$s_0 \parallel\!| s_1 \underline{\leftrightarrow} s_1 \parallel\!| s_0$: The relation $R = \{(t_0 \parallel\!| t_1, t_1 \parallel\!| t_0) \mid t_0, t_1 \in AS\}$ is a stateless bisimulation relation.

$(s_0 \parallel\!| s_1) \parallel\!| s_2 \underline{\leftrightarrow} s_0 \parallel\!| (s_1 \parallel\!| s_2)$: The symmetric closure of the relation $R = \{((t_0 \parallel\!| t_1) \parallel\!| t_2, t_0 \parallel\!| (t_1 \parallel\!| t_2)) \mid t_0, t_1, t_2 \in AS\}$ is a stateless bisimulation relation.

$rn \curvearrowright s[s] \underline{\leftrightarrow} s$: The symmetric closure of the relation $R = \{(rn \curvearrowright s[s], s)\} \cup Id$ is a stateless bisimulation relation.

$rn \curvearrowright rn[\text{skip}] \underline{\leftrightarrow} rn$: The symmetric closure of the relation $R = \{(rn \curvearrowright rn[\text{skip}], rn)\} \cup Id$ is a stateless bisimulation relation.

$(rn \curvearrowright s_0[s_1]) \ ; s_2 \underline{\leftrightarrow} rn \curvearrowright (s_0 \ ; s_2)[s_1 \ ; s_2]$: The symmetric closure of the relation $R = \{((rn \curvearrowright s_0[s_1]) \ ; s_2, rn \curvearrowright (s_0 \ ; s_2)[s_1 \ ; s_2])\} \cup Id$ is a stateless bisimulation relation.

$rn \curvearrowright (rn \curvearrowright s_0[s_1])[s_1] \underline{\leftrightarrow} rn \curvearrowright s_0[s_1]$: The symmetric closure of the relation $R = \{(rn \curvearrowright (rn \curvearrowright s_0[s_1])[s_1], rn \curvearrowright s_0[s_1])\} \cup Id$ is a stateless bisimulation relation.

$\mu x.s \underline{\leftrightarrow} s(\mu x.s/x)$: The relation $R = \{(\mu x.s, s(\mu x.s/x))\} \cup Id$ is a stateless bisimulation relation.

$\mu x.s \underline{\leftrightarrow} \mu y.s(y/x)$ if y does not occur in s: Let R be the smallest congruence such that $(\mu x.s, \mu y.s(y/x)) \in R$. Then R is a stateless bisimulation.

$s_0 \leqq s_0 + s_1$: The relation $R = \{(s_0, s_0 + s_1)\} \cup Id$ is a stateless simulation relation.

$rn \curvearrowright s_0[s_1] \leqq s_0 + s_1$: The relation $R = \{(rn \curvearrowright s_0)[s_1], s_0 + s_1)\} \cup Id$ is a stateless simulation relation.

$s_0 \parallel\!| s_1 \leqq s_0 \parallel s_1$: The relation $R = \{(t_0 \parallel\!| t_1, t_0 \parallel t_1) \mid t_0, t_1 \in AS\}$ is a stateless simulation relation.

References

Abadi, Martín and Lamport, Leslie. 1994. An Old-Fashioned Recipe for Real Time. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept.), 1543–1571.

Abrial, Jean-Raymond, Börger, Egon, and Langmaack, Hans, Editors. 1996. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. Volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany.

Aceto, L., Fokkink, W.J., and Verhoef, C. 2001. Structural Operational Semantics. In *Handbook of Process Algebra*, Bergstra, J.A., Ponse, A., and Smolka, S.A., Editors. Elsevier Science B.V., Amsterdam, chapter 3, 197–292.

Aceto, Luca and Murphy, David. 1996. Timing and Causality in Process Algebra. *Acta Informatica 33*, 4, 317–350.

Arbab, Farhad. 1996. The IWIM Model for Coordination of Concurrent Activities. In *Proceedings of the First International Conference on Coordination Models and Languages*, Volume 1061 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 34–56.

Arbab, Farhad, Baier, Christel, Boer, Frank S. de, and Rutten, Jan J. M. M. 2004. Models and Temporal Logics for Timed Component Connectors. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*. IEEE Computer Society, Los Alamitos, CA, USA, 2004, 198–207.

Arbab, Farhad and Rutten, Jan J. M. M. 2003. A Coinductive Calculus of Component Connectors. In *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, Volume 2755 of *Lecture Notes in Computer Science*. Springer, 34–55.

Balir, Lynne, Blair, Gordon, and Anderson, Anders. 1998. Separating Functional Behavior and Performance Constraints: Aspect-Oriented Specification. Tech. Report MPG-98-07, Lancaster, UK.

Banâtre, Jean-Pierre, Fradet, Pascal, and Le Métayer, Daniel. 2001. Gamma and the Chemical Reaction Model: Fifteen Years After. In *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, Calude, Cristian S., Paun, Gheorghe, Rozenberg, Grzegorz, and Salomaa, Arto, Editors. Volume 2235 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 17–44.

Banâtre, Jean-Pierre and Le Métayer, Daniel. 1993. Programming by Multiset Transformation. *Communications of the ACM 36*, 1 (Jan.), 98–111.

Bergstra, Jan A. and Klint, Paul. 1998. The discrete time ToolBus – A software coordination architecture. *Science of Computer Programming 31*, 2-3 (July), 205–229.

Börger, Egon and Gotzhein, Reinhard. 2000. The Light Control Case Study: A Synopsis. *Journal of Universal Computer Science 6*, 7, 582–586.

Brogi, Antonio and Jacquet, Jean-Marie. 2003. On the Expressiveness of Coordination via Shared Dataspaces. *Science of Computer Programming 46*, 1-2, 71–98.

Carriero, Nicholas and Gelernter, David. 1989. Linda in Context. *Communications of the ACM 32*, 4 (Apr.), 444–459.

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

Chaudron, Michel R. V. and de Jong, Edwin. 1996. Schedules for Multiset Transformer Programs. In *Coordination Programming: Mechanisms, Models and Semantics*. Imperial College Press, London, UK, 195–210.

Davies, Jim and Schneider, Steve. 1995. A brief history of Timed CSP. *Theoretical Computer Science 138*, 2 (Feb.), 243–271.

de Jong, Hayco A. and Klint, Paul. 2003. ToolBus: the Next Generation. In *Proceedings of Formal Methods Components and Objects 2002 (FMCO'02)*, Volume 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, Berling, Germany, 220–241.

Dijkstra, Edsger W. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey.

Elrad, Tzilla, Filman, Robert E., and Bader, Atef. 2001. Aspect-oriented programming - introduction. *Communications of the ACM 44*, 10, 29–32.

FREEMAN, ERIC, HUPFER, SUSANNE, AND ARNOLD, KEN. 1999. *JavaSpaces(TM) Principles, Patterns, and Practice*. Addison-Wesley.

FRØLUND, SVEND. 1992. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands, June 29 - July 3, 1992, Proceedings*, Volume 615 of *Lecture Notes in Computer Science*. Springer, 185–196.

FRØLUND, SVEND AND AGHA, GUL. 1993. A Language Framework for Multi-Object Coordination. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, Volume 707 of *Lecture Notes in Computer Science*. Springer, 346–360.

GELERNTER, DAVID AND CARRIERO, NICHOLAS. 1992. Coordination languages and their significance. *Communications of the ACM 35*, 2, 97–107.

GLABBEEK, R.J. VAN. 2001. The linear time – branching time spectrum I. The semantics of concrete, sequential processes. In *Handbook of Process Algebra*, Bergstra, J.A., Ponse, A., and Smolka, S.A., Editors. Elsevier Science B.V., Amsterdam, chapter 1, 3–99.

GLABBEEK, ROB J. VAN. 1993. The Linear Time - Branching Time Spectrum II. In *International Conference on Concurrency Theory (CONCUR'93)*, Volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1993, 66–81.

HANKIN, CHRIS L., LEMÉTAYER, DANIEL, AND SANDS, DAVID. 1993. A Calculus of Gamma Programs. In *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Machines, New Haven, Connecticut*, Volume 757 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 342–355.

HANNEMANN, ULRICH AND HOOMAN, JOZEF. 2001. Formal Design of Real-Time Components on a Shared Data Space Architecture. In *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC'01)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 143 – 150.

HARRISON, WILLIAM AND OSSHER, HAROLD. 1993. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Paepcke, Andreas, Editor. Volume 28 of *ACM SIGPLAN Notices*. ACM Press, 411–428.

HOOMAN, JOZEF AND VAN DE POL, JACO. 2005. Semantic models of a timed distributed dataspace architecture. *Theor. Comput. Sci. 331*, 2-3, 291–323.

JACQUET, JEAN-MARIE, DE BOSSCHERE, KOENRAAD, AND BROGI, ANTONIO. 2000. On Timed Coordination Languages. In *Proceedings of Coordination Languages and Models, 4th International Conference, Limassol, Cyprus*, Volume 1906 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 81–98.

LEE, EDWARD A. AND SANGIOVANNI-VINCENTELLI, ALBERTO. 1998. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17*, 12 (Dec.), 1217–1229.

MILNER, ROBIN A. 1980. *A Calculus of Communicating Systems*. Volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag.

MOUSAVI, MOHAMMADREZA, BASTEN, TWAN, RENIERS, MICHEL, CHAUDRON, MICHEL, AND RUSSELLO, GIOVANNI. 2002. Separating Functionality, Behavior and Timing in The Design of Reactive Systems: (GAMMA + Coordination) + Time. Tech. Report 02-09, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

MOUSAVI, MOHAMMADREZA, RENIERS, MICHEL, BASTEN, TWAN, AND CHAUDRON, MICHEL. 2003. Separation of Concerns in the Formal Design of Real-Time Shared Data-Space Systems. In *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD'03),*. IEEE Computer Society Press, Los Alamitos, CA, USA.

MOUSAVI, MOHAMMADREZA, RENIERS, MICHEL, BASTEN, TWAN, CHAUDRON, MICHEL, RUSSELLO, GIOVANNI, CURSARO, ANGELO, SHUKLA, SANDEEP, GUPTA, RAJESH, AND SCHMIDT, DOUGLAS C. 2002. Using Aspect-GAMMA in the Design of Embedded Systems. In *Proceedings of Seventh Annual IEEE International Workshop on High Level Design Validation and Test*. IEEE Computer Society Press, Los Alamitos, CA, USA, Cannes, France, 69–74.

MOUSAVI, MOHAMMADREZA, RENIERS, MICHEL, AND GROOTE, JAN FRISO. 2004. Congruence for SOS with Data. In *Proceedings of Nineteenth Annual IEEE Symposium on Logic in Computer Science*

(LICS'04). IEEE Computer Society Press, Turku, Finland, 302–313.

Mousavi, MohammadReza, Reniers, Michel, and Groote, Jan Friso. 2005. Notions of Bisimulation and Congruence Formats for SOS with Data. *Information and Computation (I&C) 200*, 1, 104–147.

Mousavi, MohammadReza, Reniers, Michel A., and Groote, Jan Friso. 2007. SOS formats and meta-theory: 20 years after. *Theor. Comput. Sci. 373*, 3, 238–272.

Nielsen, Brian, Ren, Shangping, and Agha, Gul. 1998. Specification of Real-Time Interaction Constraints. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 206–214.

Papadopoulos, George A. and Arbab, Farhad. 1996. Coordination of Systems With Real-Time Properties in Manifold. In *Proceedings of the Twentieth Annual International Computer Software and Applications Conference (COMPSAC'96)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 50–55.

Papadopoulos, George A. and Arbab, Farhad. 1998. Coordination models and languages. In *The Engineering of Large Systems*, Zelkowitz, Marvin, Editor. Volume 46 of *Advances in Computers*. Academic Press, The Netherlands, 330–396.

Park, David M.R. 1981. Concurrency and Automata on Infinite Sequences. In *Proceedings of 5th GI Conference*, Volume 104 of *Lecture Notes in Coputer Science*. Springer-Verlag, Berling, Germany, 167–183.

Russello, Giovanni, Chaudron, Michel R. V., and van Steen, Maarten. 2004. Exploiting Differentiated Tuple Distribution in Shared Data Spaces. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*, Volume 3149 of *Lecture Notes in Computer Science*. Springer, 579–586.

Syropoulos, Apostolos. 2001. Mathematics of Multisets. In *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, Calude, Cristian S., Paun, Gheorghe, Rozenberg, Grzegorz, and Salomaa, Arto, Editors. Volume 2235 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 347–358.

Tarr, Peri, Ossher, Harold L., Harrison, William H., and Sutton, Jr., Stanley M. 1999. N Degrees of Separation: Multi-Dimentional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*. IEEE Computer Society Press / ACM Press, 107–119.

Vereijken, Jan Joris. 1997. *Discrete Time Process Algebra*. PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

Willig, Andreas and Schieferdecker, Ina. 1996. Specifying and Verifying the Steam-Boiler Control System with Time Extended LOTOS. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler*, Abrial, Jean-Raymond, Börger, Egon, and Langmaack, Hans, Editors. Volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 473–492.