

Separation of Concerns in the Formal Design of Real-Time Shared Data-Space Systems *

MohammadReza Mousavi, Michel Reniers, Twan Basten, Michel Chaudron

Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{m.r.mousavi, m.a.reniers, a.a.basten, m.r.v.chaudron}@tue.nl

Abstract

This paper proposes a formal framework for the design of real-time shared data-space systems. The proposed method separates the concerns of functionality, behavior, and timing. This work exploits the idea of separation of concerns at the specification and design level, and it establishes a robust theoretical basis that allows rigid analysis and verification of (timed) designs.

1 Introduction

Separating different concerns in software design has been proposed in several classic computer science texts since the very beginnings of this discipline. Providing abstract and simple formalisms that are tailor-made for a single concern of requirement specification, design, or programming, seems to be of an overwhelming importance. The idea of using these tailor-made formalisms can help in a more focused design method that enables designers to concentrate on each aspect of a design separately. Furthermore, they will ease changing an aspect without being directly involved with other ones. Also, separation of concerns facilitates reuse of each aspect in other specialized 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) [7, 9] and in particular in the Aspect Oriented Programming (AOP) [7] and Multi-Dimensional Separation of Concerns [14] methods.

This paper takes a step toward this ultimate goal by providing a formal framework of separation of concerns for real-time shared data-space systems. It benefits from pre-

vious research on the functionality / behavior modelling paradigm [3] and extends it to functionality / timing / behavior. 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.

In our approach, the basic functionality of a system is designed using an abstract formal model of computation called GAMMA [2], operating on a shared data-space that is modeled by a multiset. Timing information is added to GAMMA functionalities in the form of separate intervals. Composed behavior of the system is expressed in a coordination language named *schedules*, specifying the order of basic computations using parallelism, true concurrency, synchronization, etc.

The rest of this paper is organized as follows. Section 2 introduces the preliminary concepts of our data and computation model. Section 3 defines the theory of Timed-GAMMA which serves as a model for timed functionality. Section 4 presents our coordination language for ordering the behavior of Timed-GAMMA programs. Subsequently, Section 5 uses the introduced theory to model an academic case study, a control system for a steam boiler. Section 6 presents an overview on related work, and finally, Section 7 gives concluding remarks and shows directions of ongoing research.

2 Preliminaries: Multisets and Computation

Definition 1 (Multiset) A multiset is a set that allows multiple occurrences of an element. It is defined in terms of a total function from a set of elements U (for universe) to the set of natural numbers N presenting their number of occurrence (cardinality).

We use $e \in M$ to denote that the element e has a cardinality greater than zero in multiset M . To define a multiset by

*This work is partially supported by NWO as a part of the project SACC (612.063.001).

enumerating its members (extensional presentation), we use the notation $[m_0, m_1, \dots]$, where each element is repeated as often as its cardinality. The empty multiset is denoted by \emptyset . As basic operations and relations, we use multisubset (\sqsubseteq), union (\sqcup), intersection (\sqcap), addition (\boxplus), and subtraction (\boxminus) of multisets. For a precise treatment of these operations and their properties, see [11].

In our theory, computations are modelled by rewriting multisets using a multiset of concurrent substitutions (computations).

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

Intuitively, applying a computation with a single substitution $\sigma = [N/N']$ to M should result in taking N' from M and putting back multiset N . In this operation, some parts of the multiset may be only temporarily taken away by N' and put back by N again (the *read* part). 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. We formalize this intuition by defining the *read*, *take*, and *put* parts of a computation inductively:

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

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

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

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 3 (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 $\text{read}(\sigma_0 \boxplus \sigma_1) \boxplus \text{take}(\sigma_0 \boxplus \sigma_1) \sqsubseteq M$.

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.

For the sake of brevity, we do not present details of multiset structure in this paper. We only assume that multiset elements are members of a basic set, closed under functions of a given logical structure.

Example 1 (Computations and Independence)

Consider the computations $\sigma_0 = [[0]/[0, 1]]$ and $\sigma_1 = [[0]/[0, 2], [2]/[1]]$. The first computation removes $[0, 1]$ from the multiset and places the element $[0]$ back. Thus, we call $[0]$ the *read* part and $[1]$ the *take* part of this computation. The *put* part of this computation is the empty multiset. Similarly, for the second computation, $[0]$, $[2, 1]$ and $[2]$ are the *read*, *take* and *put* parts, respectively. Note that, for the second computation, 2 is not considered in the *read* part according to Definition 2. This is in line with the intuition that the two substitutions in this computation are parallel and thus application of one should be independent of the other.

For the multiset $M_0 = [0, 1, 1, 2]$, it holds that $M_0 \models \sigma_0 \bowtie \sigma_1$. However, the independence of σ_0 and σ_1 holds for none of the two multisets $M_1 = [0, 1, 2]$ and $M_2 = [0, 1, 1]$. This fact matches the intuition behind the independence relation. For M_1 , application of σ_0 prohibits (application of) the second substitution in σ_1 . Since application of the first substitution of σ_1 to M_2 is not possible σ_0 and σ_1 are not independent with respect to M_2 .

3 Timed-GAMMA

3.1 Syntax of Timed-GAMMA

Figure 1 defines the syntax of Timed-GAMMA. A Timed-GAMMA program consists of a program (defining un-timed functionality) and timing information. A program consists of a name and a non-empty set of rules, each rewriting the content of the shared multiset. Each rule consists of a name and a (possibly empty) set of terms in the left- and right-hand side of the substitution arrow \mapsto and a condition part that are to be evaluated by multiset content. A multiset expression (*MultisetExp*) is a list of expressions built on a set of variables using the previously mentioned logical structure and conditions (*Cond*) are assumed to be propositions (containing variables) defined over the same logical structure.

Timing constraints are added to the program definition to represent an estimation of rule execution time. Such a timing estimation relates a rule to an interval representing the minimum and maximum time needed to perform a computation of this rule. This time is relative to the point from which the rule is scheduled for execution. A basic time domain *Time* with equality, an addition operation $+$ with unit element 0 , and a total ordering $<$ with least element 0 is assumed. We extend this basic time domain with an infinity

<i>Program</i>	$::=$ <i>ProgramName</i> = { <i>Rules</i> }
<i>Rules</i>	$::=$ <i>Rule</i> <i>Rule</i> , <i>Rules</i>
<i>Rule</i>	$::=$ <i>RuleName</i> = <div style="margin-left: 20px;"><i>MultisetExp</i> \mapsto <i>MultisetExp</i> \Leftarrow <i>Cond</i></div>
<i>TimedProgram</i>	$::=$ <i>Program</i> <i>Timing</i>
<i>Timing</i>	$::=$ ϵ , <i>TConstraint</i> <i>Timing</i>
<i>TConstraint</i>	$::=$ <i>TRuleName</i> = <i>Interval</i>
<i>Interval</i>	$::=$ [<i>Time</i> , <i>Time</i>] (<i>Time</i> , <i>Time</i>] <div style="margin-left: 20px;"> [<i>Time</i>, <i>Time</i>$_{\infty}$] (<i>Time</i>, <i>Time</i>$_{\infty}$)</div>

Figure 1. Abstract Syntax of Timed-GAMMA

element to obtain $Time_{\infty}$ in a standard way. We refer to the lower and upper bound of an interval I with $lb(I)$ and $ub(I)$, respectively.

Example 2 (Mutual Exclusion) To illustrate the syntax of timed-GAMMA programs, we present a program that models processes competing to enter a critical section. The system consists of n processes numbered from 1 to n . Mutual exclusion is maintained by claiming a shared token which is assumed to be some constant integer greater than n :

$$\begin{aligned}
 ME = \{ & \\
 & \textit{Enter} = \textit{Token}, i \mapsto \textit{Token} + i \Leftarrow 1 \leq i \leq n, \\
 & \textit{Leave} = \textit{Token} + i \mapsto \textit{Token}, i \Leftarrow 1 \leq i \leq n \}, \\
 & T_{\textit{Enter}} = [t_{\textit{minAcq}}, t_{\textit{maxAcq}}]
 \end{aligned}$$

The above simple program consists of two rules; rule *Enter* models a process claiming the token and entering the critical section and *Leave* models leaving the critical section and putting the token back in the multiset. Rule *Enter* is assumed to have a specific timing specification but the timing of *Leave* is left unspecified, meaning that the process can remain in the critical section for an arbitrary time.

3.2 Semantics of Timed-GAMMA

The semantics of Timed-GAMMA is presented in two parts. The first part, presented in Figure 2, shows the basic timed-computation and termination of a rule and, based on that, the second part in Figure 3 defines the behavior of Timed-GAMMA programs. In the given semantics, the state $\langle r, M, T \rangle$ consists of the rule r , the multiset M (of shared data), and the multiset of scheduled tasks T (an extension of the computation notion to timed settings). A task $\alpha@t : I$ is the substitution α together with the elapsed processing time t (the duration that the task has been active and running till now), and the estimated execution time interval I . The transitions in the given semantics are either of the

(RuleTerm)	$\frac{\neg \exists \bar{v} M, \bar{v} \propto r}{\langle r, M, \square \rangle \xrightarrow{\sqrt{1}}}$
(RuleSched)	$\frac{M, \bar{v} \propto r}{\langle r, M, \square \rangle \xrightarrow{0} \langle r, M, [\alpha@0 : r.I] \rangle}$ where for a rule $r = m_0 \mapsto m_1 \Leftarrow c, \alpha = \bar{v}[m_1]/\bar{v}[m_0]$
(RuleComp)	$\frac{t \in I}{\langle r, M, [\alpha@t : I] \rangle \xrightarrow{[\alpha]} \langle r, M([\alpha]), \square \rangle}$
(TimePass)	$\frac{t + t' \in I \vee t + t' \leq lb(I) \quad t' > 0}{\langle r, M, [\alpha@t : I] \rangle \xrightarrow{t'} \langle r, M, [\alpha@t + t' : I] \rangle}$

Figure 2. Semantics of Basic Timed Functionality

form \xrightarrow{t}_1 (where t can range over the basic time domain *Time*) or $\xrightarrow{[\alpha]}_1$, where α ranges over substitutions. A basic timed-computation is divided into three phases.

The first phase consists of scheduling a task (see rule **(RuleSched)**). This is only possible if there exists a valuation \bar{v} that both satisfies the condition and evaluates the left-hand-side expression of rule r to be a part of the multiset (denoted by $M, \bar{v} \propto r$). The scheduling of a task is indicated by a transition $\xrightarrow{0}_1$. 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. To refer to the interval I associated with rule r , we use $r.I$. If there is no interval defined for r , $r.I$ 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 rule.

Example 3 (Scheduling a Task) Consider program *ME* of Example 2 with initial multiset $M = [1, \dots, n, \textit{Token}]$. One possible scheduling of rule *Enter* is the following:

$$\begin{aligned}
 & \langle \textit{Enter}, M, \square \rangle \xrightarrow{0}_1 \langle \textit{Enter}, M, [\alpha@0 : I] \rangle, \\
 & \text{with } \alpha = [\textit{Token} + 2]/[\textit{Token}, 2] \text{ and } I = T_{\textit{Enter}}.
 \end{aligned}$$

For the time being, we assume that $r.I$ works as a function. Nevertheless, this assumption could be relaxed by allowing several intervals associated to a rule, and hence $r.I$ returning a set of time points which is not necessarily a single convex interval. This relaxed assumption would not require major change in our semantics. However, for simplicity we assume the single interval time paradigm from now on. Another useful extension is to declare the bounds of intervals as functions of system state. Because the information for determining the execution time is available at the task scheduling time, this is also a straightforward extension

to our model.

The second phase of basic computation is spending (processor) time on a computation, using rule (**TimePass**). Finally, the third phase is performing (committing) a computation via (**RuleComp**), which results in substituting the left-hand-side by the right-hand-side valuation in the multiset.

The division of 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).

Rule (**RuleTerm**) is dedicated to 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 second part of the semantics, given in Figure 3, specifies the general chaotic behavior of Timed-GAMMA programs by composing behavior of rules in all possible orders and all possible levels of true concurrency. In this semantics, the states are of the form $\langle R, M, T \rangle$ where R is a timed-GAMMA program and M and T are as before. We use \surd , \xrightarrow{t} and $\xrightarrow{\sigma}$ instead of \surd_1 , \xrightarrow{t}_1 and $\xrightarrow{[\alpha]}_1$, respectively. Rule (**ProgTerm**) extends the termination of a single rule to termination of a program. It is worth mentioning that in the context of a program, rule termination is not necessarily permanent; a terminated rule may become enabled later due to activity of other rules. Only if all rules terminate, termination becomes permanent. Rules (**ProgComp0**) and (**ProgComp1**) specify how a Timed-GAMMA program can perform computations. (**ProgTime0**) and (**ProgTime1**) specify spending time on execution of computations. The above four rules provide an abstraction from the true level of concurrency. The rule (**ProgSched**) shows that a program can schedule a new task if it can be scheduled by one of its rules and it is independent from the current context of parallel tasks (where the independence relation between two task multisets means independence of the multiset of their respective computations). The semantics of Timed-GAMMA programs 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 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).

Example 4 (Mutual Exclusion, Revisited) Consider the program in Example 2. If the program starts from the initial multiset that contains no process in the critical section with only one token, i.e., the multiset $M = [1, \dots, n, \text{Token}]$, then the abstract behavior of the program (defined in Figure 3) maintains the mutual exclusion property. This is due to the fact that, on one hand, for a multiset containing one token, any two computations resulting from rule *Enter* are not independent. On the other hand, the rule *Enter* is

disabled when there is no token in the multiset.

However, the abstract behavior of this program does not maintain desired liveness properties, preventing individual starvation, for example, not even if we specify a maximum timing for the *Leave* rule. To enforce such properties, we need to coordinate the execution of rules. We do this by defining a coordination language for rules in the next section.

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 coordinated behavior of schedules (Section 4). In other words, the first part of the semantics serves to define the atomic units of functionality. Technically speaking, one can replace this particular model of timed functionality with an operational semantics of another functionality model (say, JavaSpace 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 lose the orthogonality in the model as it is presented here.

4 Schedules

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, the aspect of coordination should be kept orthogonal w.r.t. timing to the largest possible extent. Thus, we assume a timed functionality model and use it as the basis of our coordination model. Hence, a schedule does not add explicit information about timing and functionality.

4.1 Syntax of Schedules

The syntax of our language is specified in Figure 4. *RuleName* is the notation to adopt Timed-GAMMA rules as the building blocks of a *Schedule*. The rule-conditional operator \curvearrowright is used to provide different strategies based on whether or not a rule can be scheduled. Sequential composition of schedules is denoted by $;$. Abstract parallel composition ($||$) allows for both concurrent and serialized execution of components (to represent the cases where there may or

```

Schedule ::= RuleName | Schedule ; Schedule
           | RuleName  $\curvearrowright$  Schedule [Schedule]
           | Schedule || Schedule | Schedule ||| Schedule
           |  $\mu$  RecursionVar . Schedule | RecursionVar

```

Figure 4. Syntax of Schedules

tion ($||$) allows for both concurrent and serialized execution of components (to represent the cases where there may or

$$\begin{array}{l}
\text{(ProgTerm)} \frac{\forall r \in R \langle r, M, \Box \rangle \sqrt{1}}{\langle R, M, \Box \rangle \sqrt{1}} \quad \text{(ProgComp0)} \frac{r \in R \quad \langle r, M, [\alpha @ t : I] \rangle \xrightarrow{[\alpha]}_1 \langle r, M', \Box \rangle}{\langle R, M, [\alpha @ t : I] \boxplus T \rangle \xrightarrow{[\alpha]}_1 \langle R, M', T \rangle} \\
\text{(ProgComp1)} \frac{\langle R, M, T_0 \rangle \xrightarrow{\sigma_0}_1 \langle R, M'_0, T'_0 \rangle \quad \langle R, M, T_1 \rangle \xrightarrow{\sigma_1}_1 \langle R, M'_1, T'_1 \rangle}{\langle R, M, T_0 \boxplus T_1 \rangle \xrightarrow{\sigma_0 \boxplus \sigma_1} \langle R, M(\sigma_0 \boxplus \sigma_1), T'_0 \boxplus T'_1 \rangle} \\
\text{(ProgTime0)} \frac{r \in R \quad \langle r, M, [\alpha @ t : I] \rangle \xrightarrow{t'}_1 \langle r, M, [\alpha @ t + t' : I] \rangle \quad t' > 0}{\langle R, M, [\alpha @ t : I] \boxplus T \rangle \xrightarrow{t'} \langle R, M, [\alpha @ t + t' : I] \boxplus T \rangle} \\
\text{(ProgTime1)} \frac{\langle R, M, T_0 \rangle \xrightarrow{t} \langle r, M, T'_0 \rangle \quad \langle R, M, T_1 \rangle \xrightarrow{t} \langle R, M, T'_1 \rangle \quad t > 0}{\langle R, M, T_0 \boxplus T_1 \rangle \xrightarrow{t} \langle R, M, T'_0 \boxplus T'_1 \rangle} \\
\text{(ProgSched)} \frac{r \in R \quad \langle r, M, \Box \rangle \xrightarrow{0}_1 \langle r, M, [\alpha @ 0 : I] \rangle \quad M \models [\alpha @ 0 : I] \bowtie T}{\langle R, M, T \rangle \xrightarrow{0} \langle R, M, [\alpha @ 0 : I] \boxplus T \rangle}
\end{array}$$

Figure 3. Semantics of Timed-GAMMA: Abstract Behavior of Programs

may not be enough resources for real concurrency). Strict parallel composition (\parallel) forces the participating components to run concurrently (provided that execution is possible at all components and that independence conditions are satisfied). The recursion operator μ is used to make recursive schedules ($\mu x.s(x)$) explicitly. 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 schedules. These names serve as a syntactic shorthand for the defined schedules.

In scheduling rules, there might be a need to strengthen the condition under which the rule $r = m_0 \mapsto m_1 \Leftarrow c$ is enabled with an additional condition c' . Instead of introducing a new rule $r' = m_0 \mapsto m_1 \Leftarrow c \wedge c'$ in such cases, we simply write $c' \triangleright r$ in the schedules.

Example 5 (A Schedule for Mutual Exclusion) A simple schedule for the timed-GAMMA program given in Example 2 is the following:

$Enter_j = (j = i) \triangleright Enter$
 $Leave_j = (j = i) \triangleright Leave$
 $RoundRobin = \mu X.((Enter_1 ; Leave_1) ; \dots ; (Enter_n ; Leave_n)) ; X$

This schedule guarantees progress and absence of individual starvation if an upper bound for duration of *Leave* is specified. This fact can be derived from the semantics of the next subsection.

4.2 Semantics of Schedules

Figure 5 shows the first set of semantic rules for the timed-coordination language. 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 w.r.t. parallel and sequential compositions; e.g. in a schedule like $s \parallel (s ; t)$). Hence, we also attach scheduled tasks of each coordination term to its respective syntactic expression (see rule **(CoordSched)**). Also observe that, in rule **(CoordComp)**, a rule, after committing a computation, is replaced by the rule *skip*, where *skip* is defined to be the rule $skip = \epsilon \mapsto \epsilon \Leftarrow false$, i.e. a rule that cannot be scheduled.

So, in the given semantics, the state $\langle s, M, T \rangle$ contains s as the coordination expression that is possibly augmented with scheduled tasks (substitution, timing, and interval), M is the data multiset, as before, and T is the multiset of active tasks, as in semantics of Timed-GAMMA. Note that the multiset of tasks can be derived from the annotated schedules, too, but we make it explicit in the state for readability and consistency (although we could require in all semantic rules that the annotated task sets should be the same as the tasks in T , this assumption is not necessary for consistency of our semantics). The schedule composition operators are defined as follows, where the transition $\xrightarrow{\chi}$ denotes scheduling a task, passage of time or performing a computation (χ is a variable that ranges over the time domain and computations).

Rule Conditional Rules (RC0) to (RC3) define the semantics for the rule-conditional operator. Expression $r \curvearrowright s[t]$ can schedule a task when either the condition rule r is enabled and the first argument s can schedule a task, or when r terminates (is disabled at the moment) and t can schedule a task. Obviously, it terminates when none of the above

$$\begin{array}{c}
\text{(CoordRuleTerm)} \frac{\langle r, M, \Box \rangle \surd_1}{\langle r, M, \Box \rangle \surd} \quad \text{(CoordSched)} \frac{\langle r, M, \Box \rangle \xrightarrow{0}_1 \langle r, M, [\alpha@0 : I] \rangle}{\langle r, M, T \rangle \xrightarrow{0} \langle r[\alpha@0 : I], M, [\alpha@0 : I] \rangle} \\
\text{(CoordTimePass)} \frac{\langle r, M, [\alpha@t : I] \rangle \xrightarrow{t'}_1 \langle r, M, [\alpha@t + t' : I] \rangle \quad t' > 0}{\langle r[\alpha@t : I], M, [\alpha@t : I] \rangle \xrightarrow{t'} \langle r[\alpha@t + t' : I], M, [\alpha@t + t' : I] \rangle} \\
\text{(CoordComp)} \frac{\langle r, M, [\alpha@t : I] \rangle \xrightarrow{[\alpha]}_1 \langle r, M', \Box \rangle}{\langle r[\alpha@t : I], M, [\alpha@t : I] \rangle \xrightarrow{[\alpha]} \langle skip, M', \Box \rangle}
\end{array}$$

Figure 5. Semantics of Timed-Coordination: Basic Computation and Termination

cases are possible:

$$\begin{array}{c}
\text{(RC0)} \frac{\neg(\langle r, M, T \rangle \surd) \quad \langle s, M, T \rangle \xrightarrow{0} \langle s', M, T' \rangle}{\langle r \curvearrowright s[t], M, T \rangle \xrightarrow{0} \langle s', M, T' \rangle} \quad \text{(P2)} \frac{\langle s_0, M, T_0 \rangle \xrightarrow{t} \langle s'_0, M, T'_0 \rangle \quad \langle s_1, M, T_1 \rangle \xrightarrow{t} \langle s'_1, M, T'_1 \rangle \quad t > 0}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{t} \langle s'_0 \parallel s'_1, M, T'_0 \boxplus T'_1 \rangle} \\
\text{(RC1)} \frac{\langle r, M, T \rangle \surd \quad \langle t, M, T \rangle \xrightarrow{0} \langle t', M, T' \rangle}{\langle r \curvearrowright s[t], M, T \rangle \xrightarrow{0} \langle t', M, T' \rangle} \quad \text{(P3)} \frac{\langle s_0, M, T_0 \rangle \xrightarrow{\sigma_0} \langle s'_0, M_0, T'_0 \rangle \quad \langle s_1, M, T_1 \rangle \xrightarrow{\sigma_1} \langle s'_1, M_1, T'_1 \rangle}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{\sigma_0 \boxplus \sigma_1} \langle s'_0 \parallel s'_1, M(\sigma_0 \boxplus \sigma_1), T'_0 \boxplus T'_1 \rangle} \\
\text{(RC2)} \frac{\neg(\langle r, M, T \rangle \surd) \quad \langle s, M, T \rangle \surd}{\langle r \curvearrowright s[t], M, T \rangle \surd} \quad \text{(P4)} \frac{\langle s_0, M, T \rangle \surd \quad \langle s_1, M, T \rangle \surd}{\langle s_0 \parallel s_1, M, T \rangle \surd} \\
\text{(RC3)} \frac{\langle r, M, T \rangle \surd \quad \langle t, M, T \rangle \surd}{\langle r \curvearrowright s[t], M, T \rangle \surd}
\end{array}$$

Sequential Composition

$$\begin{array}{c}
\text{(S0)} \frac{\langle s_0, M, T \rangle \xrightarrow{\chi} \langle s'_0, M', T' \rangle}{\langle s_0 ; s_1, M, T \rangle \xrightarrow{\chi} \langle s'_0 ; s_1, M', T' \rangle} \\
\text{(S1)} \frac{\langle s_0, M, T \rangle \surd \quad \langle s_1, M, T \rangle \xrightarrow{\chi} \langle s'_1, M', T' \rangle}{\langle s_0 ; s_1, M, T \rangle \xrightarrow{\chi} \langle s'_1, M', T' \rangle} \\
\text{(S2)} \frac{\langle s_0, M, T \rangle \surd \quad \langle s_1, M, T \rangle \surd}{\langle s_0 ; s_1, M, T \rangle \surd}
\end{array}$$

Abstract Parallel Composition Rules (P0) to (P3) 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 these rules assures that the task multiset remains consistent if either of the two sides schedules a new task. (P2) specifies concurrent execution of the two sides by allowing them to spend time synchronously. Rule (P3) represents concurrent commitment of tasks.

$$\begin{array}{c}
\text{(P0)} \frac{\langle s_0, M, T_0 \rangle \xrightarrow{\chi} \langle s'_0, M', T'_0 \rangle \quad M' \models T'_0 \bowtie T_1}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{\chi} \langle s'_0 \parallel s_1, M', T'_0 \boxplus T_1 \rangle} \\
\text{(P1)} \frac{\langle s_1, M, T_1 \rangle \xrightarrow{\chi} \langle s'_1, M', T'_1 \rangle \quad M' \models T_0 \bowtie T'_1}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{\chi} \langle s_0 \parallel s'_1, M', T_0 \boxplus T'_1 \rangle}
\end{array}$$

Strict Parallel Composition Strict parallelism only differs from the abstract one in that it does not allow one component to prohibit or delay the other one in execution. In other words, it models real concurrency in which composed processes perform their behavior independent of each other under some global consistency conditions. To model this type of composition, rules (P0) and (P1) are restricted to allow time passage only when one of the parties cannot perform an action. Thus, tasks are forced to spend their computation time together. We only rewrite (P0) which is decomposed into three rules (SP0), (SP01), and (SP02) here. The complete semantics of strict parallelism has three similar rules instead of (P1) and also copies of (P2) to (P4):

$$\begin{array}{c}
\text{(SP0)} \frac{\langle s_0, M, T_0 \rangle \xrightarrow{0} \langle s'_0, M, T'_0 \rangle \quad M \models T'_0 \bowtie T_1}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{0} \langle s'_0 \parallel s_1, M, T'_0 \boxplus T_1 \rangle} \\
\text{(SP01)} \frac{\langle s_0, M, T_0 \rangle \xrightarrow{\sigma} \langle s'_0, M', T'_0 \rangle}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{\sigma} \langle s'_0 \parallel s_1, M', T'_0 \boxplus T_1 \rangle} \\
\text{(SP02)} \frac{\langle s_0, M, T_0 \rangle \xrightarrow{t} \langle s'_0, M, T'_0 \rangle \quad \text{disabled}(\langle s_1, M, T_1 \rangle, T_0)}{\langle s_0 \parallel s_1, M, T_0 \boxplus T_1 \rangle \xrightarrow{t} \langle s'_0 \parallel s_1, M, T'_0 \boxplus T_1 \rangle}
\end{array}$$

In the above rule, $\text{disabled}(\langle s_1, M, T_1 \rangle, T_0)$ means that $\langle s_1, M, T_1 \rangle$ cannot perform any time transition or commitment of tasks (i.e., $T_1 = \Box$) and if it can schedule any new task, this task is not independent from T_0 .

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 $\mu x.x$):

$$\text{(R0)} \frac{\langle s(\mu x.s/x), M, T \rangle \xrightarrow{X} \langle s', M', T' \rangle}{\langle \mu x.s, M, T \rangle \xrightarrow{X} \langle s', M', T' \rangle}$$

$$\text{(R1)} \frac{\langle s(\mu x.s/x), M, T \rangle \surd}{\langle \mu x.s, M, T \rangle \surd}$$

5 Case Study: Steam-Boiler Control

Steam-boiler control is a classical example that has been used as a common case study in using different formal modeling and specification techniques [1]. We specify this problem in order to find the strengths and shortcomings of our design method. First, we explain the informal definition of the problem and then use our framework to model it. Due to space restrictions, we simplify the original definition as far as the simplifications do not reduce technical challenges.

The problem concerns controlling a boiling tank of a steam-boiler system to keep the water level in a safe area. The system comprises the following parts:

1. A tank with the maximum capacity C (in liters). This tank contains boiling water and the water level should always be in the safe area between N_0 and N_1 liters. However, due to corruptions in the sensors and valves, the water level may go outside the safe area. If the water level reaches the dangerous level (lower than M_0 or higher than M_1) and remains there for some time (5 seconds) a disaster may happen.
2. The tank is connected to a turbine with maximum steam capacity V (in liters/second). Water can be provided into the tank using a pump. The pump has maximum capacity P (in liters/second). After receiving the command, the pump will start/stop pouring water in less than 0.5 seconds (in order to balance the pressure).
3. There are sensors to measure the water level q (in liters) and steam quantity v and pump throughput p (in liters/second). It is assumed that a liter of water produces exactly one liter of steam.
4. The pump actuator is connected to the control system using a communication channel that has a minimum and maximum delay of 0.2 and 0.5 seconds.

We model three different phases of the system, namely: normal, rescue, and stop. The normal phase consists of maintaining the water in the safe range. If the water level goes beyond its dangerous limits (i.e., below M_0 or above M_1), the system should go to the stop phase. If a failure in

the water level sensor is detected (to be discussed shortly) the system goes to the rescue phase. Failure in any other sensor, pump, or communication channel should lead the system to the stop phase.

The rescue phase is concerned with estimating the water level using the dynamics of the system when the water level sensor does not work properly. Any further failure present in this phase or reaching a non-safe water level results in moving to the stop phase.

The stop phase can be activated both by a manual operator command and by automatic detection of errors and dangerous circumstances. It results in stopping the control program.

The failure of equipment can be detected if they measure some data that contradicts static or dynamic limits of the system, or if they change their state without being requested to do so (or vice versa, they do not change their state when requested to do so). In this case study, we only model failure detection of the pump and the water sensor.

5.1 The Multiset

In this subsection, we introduce the data items that are used in defining the rules in the next subsection. All data items in the multiset will be of the form $(name, value)$ where *name* is a label describing the type of information given by *value*. We present four groups of data items which together form the global multiset.

- *SystemStatus*: used to store and retrieve information about the *SystemPhase* (with possible values *normal*, *rescue*, or *stop*) and the status (*ok* or *defect*) of the pump and water sensor (*PumpSensorStatus* and *WaterSensorStatus* respectively).
- *ChannelData*: designed to indicate the state of the commands submitted to the dedicated channel between the control program and the pump. *PumpCommand* represents the command submitted to the channel (*open*, *closed*, or *none*) and *PumpState* represents the last command received by the pump (at the other end of the channel) at time *PumpChangeTime*.
- *Dynamics*: dedicated to the status of the dynamics of the system; this includes the water level (*WaterLevel*), the throughput of the pump (*PumpThroughput*), the volume of the steam (*SteamVolume*), and the system clock (*SystemTime*).
- *EstimationData*: contains the timing of the last estimation of pouring of water into the tank (*LastPoured*), the last evaporation of steam from the tank (*LastSteamOut*), and the estimation of the water level (*EstWaterLevel*).

5.2 Timed-GAMMA Program

In this part, we formalize the basic functionality model using Timed-GAMMA. We use the following abbreviations:

- The condition *true* is omitted from the rules:
 $r = m_0 \mapsto m_1 \triangleq r = m_0 \mapsto m_1 \leftarrow true.$
- The *read* part of the rules is indicated explicitly:
 $r = d_0?, \dots, d_n? : m_0 \mapsto m_1 \leftarrow c \triangleq$
 $r = d_0, \dots, d_n, m_0 \mapsto d_0, \dots, d_n, m_1 \leftarrow c.$
- We usually refer to the pair $(name, v)$ or to v itself by mentioning only the *name*, e.g., we write $r = name \mapsto v'$ as a shorthand for $r = (name, v) \mapsto (name, v')$.
- We assume the existence of a rule *idle* : $\epsilon \mapsto \epsilon$ which represents the rule doing nothing.

The first set of rules are those dedicated to modeling the *sensors*. The functionality of the sensors is to put arbitrary values for observed variables. This is to model both real sensors that report *real* and in-bound values, as well as defective sensors that report out-of-bound ones.

$$\begin{aligned} WaterSense &= WaterLevel \mapsto x, \\ PumpSense &= PumpThroughput \mapsto x, \\ SteamSense &= SteamVolume \mapsto x \end{aligned}$$

Since we do not have any information about timing of sensors we leave it unspecified (resulting in $[0, \infty)$). From now on, unless otherwise specified, we assume that the timing for rules performing some checks and computations and updating the multiset (such as estimating water level or checking for faults) is $[0.3, 0.5]$. Those rules that only update some tuples (without any computation) or perform some checks (without any update) only require $[0.2, 0.3]$ amount of time.

Issuing the commands for opening and closing the *pump* goes as follows. Issuing these commands only takes 0.1 units of time, since the command will only be submitted to the channel for transmission:

$$\begin{aligned} StartPump &= SystemTime? : \\ &PumpCommand, PumpChangeTime \\ &\mapsto open, SystemTime, \\ StopPump &= SystemTime? : \\ &PumpCommand, PumpChangeTime \\ &\mapsto closed, SystemTime, \\ T_{StartPump} &= T_{StopPump} = [0.1, 0.1] \end{aligned}$$

The Timed-GAMMA rules for the *channel* consist of transporting the data from one side of the channel to the other side. We simplify the model by putting the timing of transporting the command and the pump reaction together, so that we do not need to model the pump action separately:

$$\begin{aligned} Transport &= PumpCommand, PumpState \\ &\mapsto none, PumpCommand \\ &\leftarrow PumpCommand = none, \\ T_{Transport} &= [0.2, 1] \end{aligned}$$

The *estimation* rules are responsible for estimating the water level by adding poured water and subtracting evaporated steam. The *Synchronize* rule resets the estimated level to the measured level.

$$\begin{aligned} PourWater &= PumpThroughput?, SystemTime? : \\ &EstWaterLevel, LastPoured \mapsto \\ &EstWaterLevel + WaterPoured, SystemTime, \\ EmptySteam &= SteamVolume?, SystemTime? : \\ &EstWaterLevel, LastSteamOut \mapsto \\ &EstWaterLevel - SteamOut, SystemTime, \\ Synchronize &= WaterLevel?, SystemTime? : \\ &EstWaterLevel, LastPoured, LastSteamOut \\ &\mapsto WaterLevel, SystemTime, SystemTime \\ &\leftarrow (WaterLevel < C) \end{aligned}$$

In the rules, *WaterPoured* and *SteamOut* are short-hands for the following expressions.

$$\begin{aligned} WaterPoured &\triangleq (PumpThroughput * \\ &(SystemTime - LastPoured)) \\ SteamOut &\triangleq (SteamVolume * \\ &(SystemTime - LastSteamOut)) \end{aligned}$$

Fault detection checks for out-of-bound and inconsistent values to mark defective devices. The pump is assumed to be defective if it does not react to the issued command after specified delay D :

$$\begin{aligned} WaterSensorError &= WaterLevel? : \\ &WaterSensorStatus \mapsto defect \\ &\leftarrow (WaterLevel > C), \\ PumpSensorError &= PumpState?, PumpThroughput?, \\ &SystemTime?, PumpChangeTime? : \\ &PumpSensorStatus \mapsto defect \\ &\leftarrow (PumpState = closed \\ &\wedge PumpThroughput \neq 0) \\ &\vee (PumpState = open \\ &\wedge PumpThroughput = 0 \\ &\wedge SystemTime > PumpChangeTime + D) \\ RestorePumpSensor &= PumpSensorStatus \mapsto ok, \\ RestoreWaterSensor &= WaterSensorStatus \mapsto ok \end{aligned}$$

The *phase change* rules change the variable denoting the system phase and define the initialization process. *StopCheck* looks for the stop phase element in the multiset (we assume that user intervention results in a stop phase element in the multiset):

$$\begin{aligned} StopPhase &= SystemPhase \mapsto stop, \\ RescuePhase &= SystemPhase \mapsto rescue, \\ NormalPhase &= SystemPhase \mapsto normal, \\ StopCheck &= (SystemPhase, stop)? : idle, \\ Initialize &= PumpState, PumpSensorStatus, \\ &WaterSensorStatus, WaterLevel, \\ &EstWaterLevel, SteamVolume, SystemTime \\ &\mapsto closed, ok, ok, x, x, x, 0 \\ &\leftarrow (x > N_0) \wedge (x < N_1) \end{aligned}$$

We design a global system clock as follows:

$$\begin{aligned} Tick &= SystemTime \mapsto SystemTime + tickTime, \\ T_{Tick} &= [tickTime, tickTime] \end{aligned}$$

5.3 Coordination

The coordination part composes the GAMMA rules of the previous subsection in the desired order. We use the following abbreviations:

- For schedules s , we write $c \triangleright s$ to represent $(c \triangleright idle) \curvearrowright s[skip]$.
- Most of the time, checking the feasibility of scheduling a rule is followed by executing it. We define the following abbreviation to represent this pattern:
 $r \rightarrow s[t] \stackrel{\Delta}{=} r \curvearrowright (r ; s)[t]$.

The external environments of the steam-boiler system are the system clock and the sensors. Also, the channel performs its functionality independent of the control software structure. Since the system clock's timing should not be interfered by the timing of other components in the system, it is composed with the rest of the system using strict parallelism (since sensors do not have timing specifications, abstract parallelism works for them as well). Schedule *SystemClock* represents the system clock that executes tick actions sequentially (according to its specified timing). Schedule *Channel* is in charge of transporting data in the channel and time-stamping the receipt time.

$$\begin{aligned} SystemClock &= \mu X. Tick ; X \\ WaterSensor &= \mu X. WaterSense ; X \\ SteamSensor &= \mu X. SteamSense ; X \\ PumpSensor &= \mu X. PumpSense ; X \\ Channel &= \mu X. ((Transport ; TimeStamp) ; X) \\ Env &= SystemClock ||| WaterSensor ||| \\ &\quad SteamSensor ||| PumpSensor ||| Channel \end{aligned}$$

The schedule *ControlWater* is responsible for opening and closing the pump to keep the water in the safe range. *RescueControl* does the same job using the estimated water level. We start taking measures at N'_0 (with $N'_0 > N_0$) and N'_1 (with $N'_1 < N_1$) in the rescue phase because of imprecision induced by the estimations. The schedule *Control* combines the two control strategies in parallel:

$$\begin{aligned} ControlWater &= ((WaterLevel < N_0) \triangleright StartPump) || \\ &\quad ((WaterLevel > N_1) \triangleright StopPump) \\ WaterEstimate &= PourWater || EmptySteam \\ RescueControl &= ((EstWaterLevel < N'_0) \triangleright StartPump) || \\ &\quad ((EstWaterLevel > N'_1) \triangleright StopPump) \\ Rescue &= ((SystemPhase = rescue) \triangleright WaterEstimate) ; \\ &\quad ((SystemPhase = rescue) \triangleright RescueControl) \\ Normal &= ((SystemPhase = normal) \triangleright ControlWater) || \\ &\quad ((SystemPhase = normal) \triangleright Synchronize) \\ Control &= Normal || Rescue \end{aligned}$$

ChangePhase performs the selection of the right phase of the system (based on the state of the sensors):

$$\begin{aligned} ChangePhase &= PumpSensorError \rightarrow StopPhase \\ &\quad [WaterSensorError \rightarrow \\ &\quad (RestoreWaterSensor || RescuePhase) \\ &\quad [NormalPhase]] \\ RescueDanger &= (SystemPhase = rescue) \wedge \\ &\quad ((EstWaterLevel < M_0) \vee \\ &\quad (EstWaterLevel > M_1)) \triangleright StopPhase \\ NormalDanger &= (SystemPhase = normal) \wedge \\ &\quad ((WaterLevel < M_0) \vee \\ &\quad (WaterLevel > M_1)) \triangleright StopPhase \\ DangerCheck &= RescueDanger || NormalDanger \end{aligned}$$

The routine procedure of the control software consists of performing the control phases iteratively, as long as the stop command and the dangerous condition are not detected:

$$\begin{aligned} SteamBoiler &= Initialize ; \\ &\quad (Env ||| (\mu X. StopCheck \rightarrow skip[(DangerCheck || \\ &\quad (ChangePhase ; Control)) ; X])) \end{aligned}$$

5.4 Reasoning About the Design

Our design has the following properties:

1. The *Control* schedule takes at most 1.2 units of time. This value results from summing up the maximum execution times of the individual rules and assuming abstract parallel schedules to be sequential. Note that the two parallel schedules *Rescue* and *Normal* exclude each other since during control the phases do not change. Following the same calculations, the *StopCheck*, *DangerCheck* and *ChangePhase* schedules take at most 0.3, 0.5 and 1.1 time units, respectively. Hence, assuming that the system goes beyond its limits right after executing *StopCheck* and remains there during the *Control* and *ChangePhase* schedules, the system will be stopped after $0.3 + 0.5 + 1.1 + 1.2 = 3.1$ units of time. For the normal phase, this guarantees safety of the system.
2. The pumps will start/stop at most by $3.1 + 1 = 4.1$ units of time (considering the timing of the control loop and the communication delay together with the pump reaction time). This means that if the water level reaches its limits of N_0 or N_1 , it will take the appropriate action in stopping and starting pumps within this time. Again, in the normal phase, this can guarantee the liveness of the system; that is, if the pumps and steam turbine are able to return the system within the remaining 0.9 unit of time to the area between M_0 and M_1 then the system will not stop due to the danger of explosion.

3. Assuming that the rescue phase lasts at most t time units, our estimated water level may deviate from the real level at most by $(P+V)*t$ since pump throughput and steam volume can be at most P and V units off during this period. Note that this is a very pessimistic upper bound for the estimation error and if we exploit the upper bounds for the gradients of the dynamics this upper bound can be refined. Using this upper bound for the error in estimation, it follows that one of the following cases should hold for the rescue phase:

- (a) The duration of rescue phase should be so small that the system proceeds with normal control and stops there before the explosion occurs: $t < 5 - 4.1 = 0.9$.
- (b) Suppose that the property in item 2 holds for normal phase. This means that if the system finds the actual water level out of the interval $[N_0, N_1]$, then it is able to bring it to the safe limits in the remaining time before the system is in danger. Thus, if we start taking measures earlier in the rescue phase, such that $N'_0 - N_0 < (P + V) * t$ and $N_1 - N'_1 < (P + V) * t$, then we are able to compensate our errors in estimation. (A similar approach could be taken if we designed the *RescueDanger* schedule to stop earlier rather than at M_0 and M_1 .)

6 Related Work

We can categorize related work in the field in two parts: First, the attempts to formalize real-time functionality in shared data-space systems, and second, those works concentrating on formalizing the real-time behavior (composed functionality) of processes (namely, timed process algebras):

- 1. Models of shared data-space with time: A few attempts have been made to extend data coordination languages with time. In [10], four different timed extensions of the concurrent Linda coordination model are presented (namely, with relative delay, absolute wait, relative durational primitives, and absolute durational primitives). Although relative durational primitives are conceptually close to our extension of GAMMA rules, they are reasonably different from ours in that they force time-stamping the data-space and only allow introduction and consumption of temporary elements to/from the data-space. Furthermore, time transitions are synchronized among all durational primitives. Also, in [8] 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 did not

require time-stamping in our framework since it is not necessary when only timing information of tasks suffices for the specification of the system.

- 2. Timed Process Algebras: There are several extensions of process calculi with timing in the literature. An overview of such extensions can be found in [15, 5]. Our schedules language can be categorized as a process calculus with relative intervals with delayable actions. CCSiT is an extension of CCS, defined in [4], which is comparable to our schedules. In CCSiT, intervals are attached initially to the process terms, and they are interpreted as the execution time of enabled actions; when an action is enabled, its execution time (named as idling transition) starts. In our view, keeping timing concerns separate from causal relations of actions is helpful, especially in early stages of the design where the details of the actual implementation domain are not known. Apart from this fact, there is no notion of abstract parallelism (with the possibility of asynchronous time passage) defined in the above timed process calculi and time transitions proceed synchronously. In our view, abstract parallel composition can be useful in high-level design where the true concurrency level and resource scheduling policy is not determined yet.

Among the attempts to formalize the steam-boiler case study, those using process algebraic formalisms are close to our approach. For example, [16] uses Time Extended LOTOS to formalize the 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 the fact 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 approach tries to provide a monolithic framework that supports the above aspect specifications.

7 Conclusion and Future Directions

In this paper, we presented a framework for formal design of real-time shared data-space systems. The distinguishing feature of this framework is that it supports separation of functionality, timing and coordination as different aspects of system design. Semantics of each meaningful combination of these aspects should take care of their reflections and interactions and weave them together. In this paper, we presented the semantics of (timed) functionality and (timed) functionality plus coordination. In [11], an untimed semantics of GAMMA can be found. Apart from realizing this design philosophy, we can summarize the contributions of this paper as follows:

1. Defining and formalizing *read* parallelism in GAMMA semantics using computations and an independence relation. (The original GAMMA semantics does not have a notion of true concurrency, and thus does not contain any notion of independence.) This contribution can be seen as a formalization of the transaction-based programming of [6].
2. Extension of GAMMA and schedule semantics to timed semantics.
3. Reflection of three phases of basic transactions (scheduling, computing, and commitment) in the formal semantics of Timed-GAMMA and schedules.

Ongoing and future steps of this research include mechanization of a verification process, extension to distribution, mobility, and resource scheduling aspects, and providing transformations to the implementation domain. A mechanization of Timed-GAMMA and schedules is being undertaken using the PVS [13] proof mechanization tool. In [12], we defined basic ideas of the distribution aspect and applied it to a small case study. A middleware support for separation of different aspect is being developed. Currently, the implemented middleware supports the distribution aspect but we are planning to extend it to support the timing aspect as well.

References

- [1] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. LNCS 1165. Springer, 1996.
- [2] J.-P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In C. S. Calude, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, LNCS 2235, pages 17–44. Springer, 2001.
- [3] M. R. V. Chaudron. *Separating Computation and Coordination in the Design of Parallel and Distributed Programs*. PhD thesis, Department of Computer Science, Rijksuniversiteit Leiden, Leiden, The Netherlands, 1998.
- [4] M. Daniels. Modelling real-time behavior with an interval time calculus. In J. Vytupil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, Second International Symposium, Proceedings*, LNCS 571, pages 53–71. Springer, 1991.
- [5] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.
- [6] E. de Jong. *Transaction-based Programming*. PhD thesis, Department of Computer Science, Rijksuniversiteit Leiden, Leiden, The Netherlands, 1992.
- [7] T. Elrad, R. E. Filman, and A. Bader, editors. Communications of the ACM. Special Issue on Aspect Oriented Programming. 44(10), 2001.
- [8] U. Hannemann and J. Hooman. Formal design of real-time components on a shared data space architecture. In *Proceedings of the Annual International Computer Software and Applications Conference (COMP-SAC 2001)*, pages 143–150. IEEE Computer Society Press, 2001.
- [9] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, in ACM SIGPLAN Notices, 28(10), pages 411–428. ACM Press, 1993.
- [10] J.-M. Jacquet, K. de Bosschere, and A. Brogi. On timed coordination languages. In A. Porto and G.-C. Roman, editors, *Fourth International Conference on Coordination Models and Languages*, LNCS 1906, pages 81–99. Springer, 2002.
- [11] M. Mousavi, T. Basten, M. Reniers, M. Chaudron, and G. Russello. Separating functionality, behavior and timing in the design of reactive systems: (GAMMA + coordination) + time. Technical Report CSR-02-09, Eindhoven University of Technology, Eindhoven, The Netherlands, 2002.
- [12] M. Mousavi, G. Russello, M. Chaudron, M. Reniers, T. Basten, A. Cursaro, S. Shukla, R. Gupta, and D. C. Schmidt. Using Aspect-GAMMA in the design of embedded systems. In *Proceedings of the Seventh Annual IEEE International Workshop on High Level Design Validation and Test*. IEEE Computer Society Press, 2002.
- [13] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, LNAI 607, pages 748–752. Springer, 1992.
- [14] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press / ACM Press, 1999.
- [15] J. J. Vereijken. *Discrete Time Process Algebra*. PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1997.
- [16] A. Willig and I. Schieferdecker. Specifying and verifying the steam-boiler control system with Time Extended LOTOS. In [1], pages 473–492, Springer, 1996.