Separating functionality, behavior and timing in the design of reactive systems: (GAMMA + coordination) + time

Citation for published version (APA):

Document status and date:
Published: 01/01/2002

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Separating Functionality, Behavior and Timing in the Design of Reactive Systems: (GAMMA + Coordination) + Time\textsuperscript{1}

MohammadReza Mousavi, Twan Basten, Michel Reniers, Michel Chaudron, Giovanni Russello

\textsuperscript{1}This work is partially supported by NWO (The Netherlands Organization for Scientific Research) as a part of the project SACC.
Abstract

This report addresses the issue of separation of concerns in software architecture modelling of real-time reactive systems. First, the idea of modelling functionalities and behavior with GAMMA and its coordination language is reviewed, and a method for reasoning on GAMMA and coordinated specifications is presented. Then, the new aspect for timing is added to the design method and the extended syntax, semantics, and reasoning method are proposed. Before extending the formalisms with timing, decision points are outlined and different alternatives are investigated.
# Contents

1 Introduction ........................................... 2

2 GAMMA, Coordination, and Reasoning ................. 5
   2.1 Introduction ........................................ 5
   2.2 Multisets ........................................... 7
   2.3 GAMMA ............................................. 15
      2.3.1 GAMMA Shared Multiset .......................... 15
      2.3.2 GAMMA Syntax .................................. 16
      2.3.3 GAMMA Semantics ............................... 17
   2.4 Coordination ........................................ 23
      2.4.1 Coordination Language Syntax .................. 23
      2.4.2 Coordination Language Semantics ............... 24
      2.4.3 Equivalence of Schedules ....................... 27
      2.4.4 Local Termination .............................. 29
   2.5 Reasoning ........................................... 30

3 Time .................................................... 34
   3.1 Introduction ........................................ 34
   3.2 Modelling Time: Decision Points ................... 35
   3.3 Decisions and Their Rationale ....................... 37

4 Timed-GAMMA, Coordination, and Reasoning .......... 40
   4.1 Introduction ........................................ 40
   4.2 Timed-GAMMA ....................................... 40
      4.2.1 Timed-GAMMA Syntax ............................ 40
      4.2.2 Timed-GAMMA Semantics ....................... 42
   4.3 Timed-Coordination .................................. 52
   4.4 Urgency Operator ................................... 59
   4.5 Reasoning ........................................... 59

5 Conclusion and Research Directions .................. 64

A Proofs of Untimed Properties ......................... 70

B Proofs of Timed Properties ............................ 89
Chapter 1

Introduction

Separating different concerns in software design has been proposed in several classic computer science texts since the very beginnings of this discipline. However, this issue did not receive enough attention compared to that of integrated and powerful programming languages/formalisms that are able to cover all different concerns of requirements and design. Furthermore, rarely was there a way of distinction of or projection on a particular aspect or concern in these rich languages.

Providing abstract and simple formalisms that are tailor-made for a single aspect 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 change of each aspect without being directly involved with other ones and facilitates reuse of each aspect in other specialized designs. Thus, a new trend appeared recently 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) [13, 16] and in particular in the Aspect Oriented Programming (AOP) [13] and Multi-Dimensional Separation of Concerns [27] methods.

The ultimate goal of the research commenced by this report is to have a set of declarative and abstract specification languages for each aspect and then weave any meaningful combination of aspect designs to reflect inter-connections of aspects and move toward a more restricted model of behavior (i.e., towards an implementation). The combination of aspect weaving procedures enables local verification of properties. After a correctness analysis of the set of designs, an executable behavioral model will be derived from it. This model proposes a specific composition of basic building blocks. The derivation should preserve the proved correctness of a design or achieve it in a set of stepwise transformations. In addition to the executable model of behavior, a set of monitoring components might be generated from a specification so that non-functional aspects of specification can be checked at run-time, too.
This report takes the first steps toward this ultimate goal by providing a formal framework of separation of concerns for timed systems. It benefits from previous research on the functionality / behavior modelling paradigm [8] and extends it to functionality / timing / behavior. Figure 1.1 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 designs.

The basic functionality of a design is specified in an abstract formal language called GAMMA [5], and timing information is added to GAMMA functionalities in the form of separate intervals. Composed behavior of the system is expressed in a kind of coordination language. The correctness criteria for the composed timed behavior are then specified in a timed temporal logic. The composed behavior of the weaved functionality and timing that should meet certain correctness criteria is represented by a (timed) coordination language specifying the order of functionalities, parallelism, synchronization, etc... .

To keep these aspects separate in our framework, we provide semantics for each meaningful composition of models so that change or even absence of one model does not prevent reasoning about the other ones. In other words, design concerns, change, and reasoning can be localized by using the proposed framework. Since the design languages have a formal semantics, reasoning about the properties of a design in these languages can be done using formal verification techniques. We present examples of formal reasoning in each part. In the
practice of adding time to our design method, we enumerate the main decision points and provide reasons why a decision is made and in which cases the decision remains sound. We try to preserve the distinguishing feature of separation of concerns and orthogonality throughout the extension.

The rest of this report is organized as follows. Chapter 2 begins with an overview of GAMMA, its shared data space model (multiset), syntax and semantics. The discussion continues with presenting principles of our intention for using GAMMA, namely its inherent abstract nature, coordination of GAMMA programs, and a method for reasoning about them. These aspects motivate why this formal model is chosen to be enriched with timing information, and what are the intended applications of the new timed-GAMMA and coordination framework, and the goals aimed at by doing so. In Chapter 3, properties of time and its possible ways of representation and semantic modelling are studied. A set of decisions made in Chapter 3, results in an extended syntax, semantics, and reasoning method for timed-GAMMA and coordination in Chapter 4. Finally, Chapter 5 ends the report by presenting conclusions and future research directions. To illustrate the discussion, simple examples are presented in each section and worked on during the discussion to show different aspects of specification in our approach.

In order to take details of a real-time embedded system design into account, adding some other aspects of design such as hardware (processor) resources, and a data distribution model is important. We try to abstract from these aspects of design in this report and when necessary, we make some assumption about them. However, a complete model will be a result of weaving several aspect specifications including those mentioned above.
Chapter 2

GAMMA, Coordination, and Reasoning

2.1 Introduction

The Shared Data Space paradigm is proposed to be a means for temporal and spatial decoupling of component interactions. As shown in Figure 2.1, components can access the data space as the shared communication medium independent from each other. GAMMA (General Abstract Model for Multiset Manipulation) is a specification/programming model based on this paradigm. In GAMMA, the notion of a multiset (bag) is used to model the basic shared data space and rules represent the functionality of components [6, 5].

Due to the abstract nature of GAMMA, operations (rules) defined on the multiset are kept as independent as possible and hence unnecessary sequential relations (e.g., immense use of semi-colon notation even in parallel versions of classic programming languages [12]) is not imposed on GAMMA programs. In

![Figure 2.1: Shared Data Space Model](image)
other words, GAMMA programs can capture logically inherent parallelism in the problem definition [6]. Furthermore, GAMMA programs abstract from many other aspects such as distribution, replication, and fault tolerance.

This abstractness makes GAMMA a suitable choice for the specification of computational functionalities of software components. Using this model allows component designers to concentrate on discrete functionalities of components 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). This approach has been studied in [8], and resulted in a method for designing concurrent distributed systems.

Considering the complex nature of distributed real-time system design, extending the above mentioned design method to real-time systems may divide this complex process into designing individual (real-time) component functionalities, specification of real-time properties and requirements, and finally finding a composed behavior that uses functionalities in order to satisfy correctness requirements.

In order to establish a basis for such a real-time extension, we review the design method in the untimed setting and make changes to it to fit our intended purposes. Contributions of this report to the (untimed) GAMMA / coordination model are as follows:

1. The syntax of GAMMA is simplified by keeping only the basic functionality (rule) part of programs and postponing all structuring and control decisions to defining an appropriate coordination schedule for a program. Hence we eliminate structuring techniques like tropes in [15] and composition operators in [8].

2. In the foundation of our GAMMA model, we propose a more general and liberal notion of independent parallel tasks compared to that of [8] (Original representations of GAMMA formalism [6, 15] does not have a notion of independence) that allows more parallelism. (We present the differences in detail in Section 2.2.)

3. We use strong bisimilarity to relate programs and coordination expressions. We re-define the Most General Schedule of a program and prove it equal to the GAMMA program. An important advantage is that our new notion of the Most General Schedule has a compositional structure.

In this chapter, in Section 2.2, first we present a basic theory of multisets as the foundation for the GAMMA model and prove the properties that we desire from this basic model. After that, in Section 2.3, we give an overview on the GAMMA language and its chaotic execution model. A coordination language is defined subsequently in Section 2.4, to organize composition of basic GAMMA rules. After introduction of the coordination language, the equivalence notion on GAMMA programs and coordination expressions is defined. Finally, a way of reasoning is sketched in Section 2.5, to show how a coordinated program is
verified against desired properties expressed in Linear-time Temporal Logic [9]. As mentioned before, each section works on the corresponding aspect of given examples.

2.2 Multisets

In this subsection, we define a concise and basic theory of multisets that is to be used through the rest of the paper. For a more detailed discussion on multisets and some historical accounts, see [26].

**Definition 2.1 (Multiset)** A multiset is a set that allows multiple occurrences of elements. More precisely, it is defined in terms of a total function from a set of data objects $U$ (for universe) to the set of natural non-negative numbers $\mathbb{N}$ presenting their number of occurrence. Hence, a multiset $M$ is defined as:

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

We refer to the set of all multisets of a universe $U$ as $\mathcal{M}(U)$.

The multiset function when applied to an element from the universe, gives the cardinality of the element. If an element is not present in a multiset, the result of the multiset function (its cardinality) is zero.

**Definition 2.2 (Membership)** By definition, for all elements $e \in U$, $e$ is a member of a multiset $M \in \mathcal{M}(U)$ if its cardinality is greater than zero:

$$e \in M \overset{\Delta}{=} M(e) > 0.$$ 

If an element $e$ is not a member of a multiset $M$, this fact is denoted by $e \not\in M$.

To define a multiset by enumerating its members (extensional presentation), we use the notation $[m_1, m_2, \ldots]$, where each element is repeated as often as its cardinality. The empty multiset (denoted by $\emptyset$) is the multiset that contains no element.

**Definition 2.3 (Multisubset)** The multisubset relation is, for all $M_1, M_2 \in \mathcal{M}(U)$, defined as:

$$M_1 \subseteq M_2 \overset{\Delta}{=} \forall_{e \in U} \ M_1(e) \leq M_2(e).$$

**Definition 2.4 (Multiset Basic Operations)** To express our formal semantics, we define the following basic operations on multisets, for all $e \in U, M_1, M_2 \in \mathcal{M}(U)$:

1. Multiset union: $(M_1 \cup M_2)(e) \overset{\Delta}{=} \max(M_1(e), M_2(e))$

2. Multiset intersection: $(M_1 \cap M_2)(e) \overset{\Delta}{=} \min(M_1(e), M_2(e))$
3. Multiset addition: \((M_1 \uplus M_2)(e) \overset{\Delta}{=} M_1(e) + M_2(e)\)

4. Multiset subtraction: \((M_1 \ominus M_2)(e) \overset{\Delta}{=} \max(0, M_1(e) - M_2(e))\)

**Corollary 2.1** The following properties of basic operations hold for all multisets \(M_1, M_2\) and \(M_3 \in \mathcal{M}(U)\) [26]:

1. Commutativity:
   \[
   M_1 \cup M_2 = M_2 \cup M_1 \\
   M_1 \cap M_2 = M_2 \cap M_1 \\
   M_1 \ominus M_2 = M_2 \ominus M_1
   \]

2. Associativity:
   \[
   M_1 \cup (M_2 \cup M_3) = (M_1 \cup M_2) \cup M_3 \\
   M_1 \cap (M_2 \cap M_3) = (M_1 \cap M_2) \cap M_3 \\
   M_1 \ominus (M_2 \ominus M_3) = (M_1 \ominus M_2) \ominus M_3
   \]

3. Zero Element:
   \[
   M_1 \cup \emptyset = M_1 \\
   M_1 \ominus \emptyset = M_1 \\
   M_1 \oplus \emptyset = M_1
   \]

4. Idempotency:
   \[
   M_1 \cup M_1 = M_1 \\
   M_1 \cap M_1 = M_1
   \]

Substitutions are basic notions of computation in GAMMA semantics. To define this notion formally, first we define single substitution to represent basic computation and afterwards, we define multiple substitution as composition of basic computations.

**Definition 2.5 (Single substitution)** Let \(N\) and \(N'\) be arbitrary multiset in \(\mathcal{M}(U)\), then a single substitution is denoted by \(N/N'\), and its read, take, and put part are defined as follows:

\[
\text{read}(N/N') \overset{\Delta}{=} N \cap N' \\
\text{take}(N/N') \overset{\Delta}{=} N' \ominus \text{read}(N/N') \\
\text{put}(N/N') \overset{\Delta}{=} N \ominus \text{read}(N/N')
\]

Henceforth, applying an arbitrary single substitution \(\sigma\) on a multiset \(M \in \mathcal{M}(U)\) is defined as:

\[
M[\sigma] \overset{\Delta}{=} \begin{cases} 
(M \ominus \text{take}(\sigma)) \uplus \text{put}(\sigma) & \text{if } \text{read}(\sigma) \ominus \text{take}(\sigma) \subseteq M \\
M & \text{otherwise}
\end{cases}
\]
Intuitively, applying a single substitution 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). According to the above definition, we only remove the take part of the substitution and replace it by the put part. The distinction of these parts turns out to be useful in the remainder. Figure 2.2 represents the above definition using a Venn diagram. Also, the definition of single substitution application states that a substitution does not change a multiset if the sum of its read and take parts are not present in the multiset.

**Proposition 2.2** For an arbitrary single substitution $\sigma$: $\text{take}(\sigma) \cap \text{put}(\sigma) = \emptyset$.

**Proof.** Suppose that $\sigma = N/N'$; for an arbitrary element $e \in U$, suppose that $N(e) = n$ and $N'(e) = n'$. Assume, $n \leq n'$. Then, according to Definitions 2.4 and 2.5: $\text{put}(\sigma)(e) = (N \oplus (N' \cap N))(e) = N(e) - \min(N(e), N'(e)) = N(e) - N'(e) = 0$.

Similarly, if $n' \leq n$, $\text{take}(\sigma)(e) = 0$. So in both cases, $(\text{put}(\sigma) \cap \text{take}(\sigma))(e) = \min(\text{put}(\sigma)(e), \text{take}(\sigma)(e)) = 0$, which means that $e \not\in \text{take}(\sigma) \cap \text{put}(\sigma)$.

**Definition 2.6 (Multiple substitution)** Multiple substitution is denoted by $\sigma_1, \sigma_2$ where $\sigma_1$ and $\sigma_2$ are single or multiple substitutions. Functions $\text{take}$, $\text{put}$, and read of a multiple substitution are defined inductively, based on those for a single substitution, as follows:

- $\text{read}(\sigma_1, \sigma_2) \triangleq \text{read}(\sigma_1) \cup \text{read}(\sigma_2)$
- $\text{take}(\sigma_1, \sigma_2) \triangleq \text{take}(\sigma_1) \cup \text{take}(\sigma_2)$
- $\text{put}(\sigma_1, \sigma_2) \triangleq \text{put}(\sigma_1) \cup \text{put}(\sigma_2)$

The read part of a multiple substitution is the union of its constituent parts because a single copy of an element can be read by several substitution operations and remains unchanged afterwards. However, for the take or put parts,
different copies of the elements are removed or added by the individual substitutions. Application of a multiple substitution to a multiset is defined in exactly the same way as the application of a single substitution (see Definition 2.5).

We define that the independence condition holds for two substitutions with respect to a multiset if and only if both substitutions can be applied to the multiset simultaneously or in any arbitrary order. This is a useful notion to model parallelism in GAMMA semantics. This definition is formalized as follows:

**Definition 2.7 (Independent substitutions)** Two arbitrary (single or multiple) substitutions \( \sigma_1 \) and \( \sigma_2 \) are defined to be independent with respect to a multiset \( M \), denoted by \( M \models \sigma_1 \bowtie \sigma_2 \), as:

\[
M \models \sigma_1 \bowtie \sigma_2 \overset{\triangle}{=} \text{read}(\sigma_1, \sigma_2) \sqcup \text{take}(\sigma_1, \sigma_2) \subseteq M
\]

This means that two substitutions are independent if and only if both can find enough shared copies of elements to read and enough different copies of elements to take, or in other words, if and only if the multiple substitution \( \sigma_1, \sigma_2 \) is applicable to \( M \).

**Proposition 2.3 (Commutativity of Independence)** For all multisets \( M \) and substitutions \( \sigma_1, \sigma_2 \), \( M \models \sigma_1 \bowtie \sigma_2 \) if and only if \( M \models \sigma_2 \bowtie \sigma_1 \).

**Proof.** According to Definition 2.6 and Corollary 2.1:

\[
M \models \sigma_1 \bowtie \sigma_2 \\
\iff \text{read}(\sigma_1, \sigma_2) \sqcup \text{take}(\sigma_1, \sigma_2) \subseteq M \\
\iff (\text{read}(\sigma_1) \sqcup \text{read}(\sigma_2)) \sqcup (\text{take}(\sigma_1) \sqcup \text{take}(\sigma_2)) \subseteq M \\
\iff (\text{read}(\sigma_2) \sqcup \text{read}(\sigma_1)) \sqcup (\text{take}(\sigma_2) \sqcup \text{take}(\sigma_1)) \subseteq M \\
\iff \text{read}(\sigma_2, \sigma_1) \sqcup \text{take}(\sigma_2, \sigma_1) \subseteq M \\
\iff M \models \sigma_2 \bowtie \sigma_1
\]

\( \Box \)

**Example 2.1 (Independency of substitutions)** Consider the multiset \( M = [1,2,2] \). Two substitutions \( \sigma_1 = [2]/[1,2] \) and \( \sigma_2 = [2]/[2,2] \) are independent with respect to \( M \), because \( \text{read}(\sigma_1, \sigma_2) = [2] \sqcup [2] = [2] \) and \( \text{take}(\sigma_1, \sigma_2) = [1] \sqcup [2] = [1,2] \) and thus, \( \text{read}(\sigma_1, \sigma_2) \sqcup \text{take}(\sigma_1, \sigma_2) = [1,2,2] \subseteq M \). As we prove later, it allows both substitutions to be applied to \( M \) simultaneously (by taking \( \text{take}(\sigma_1, \sigma_2) \) from \( M \) and putting \( \text{put}(\sigma_1, \sigma_2) \)) and also in both possible orders.

However, two substitutions \( \sigma_1 = [1]/[1,2] \) and \( \sigma_2 = [2]/[2,2] \) are not independent substitutions with respect to the multiset \( M \), because \( \text{read}(\sigma_1, \sigma_2) = [1] \sqcup [2] = [1,2] \) and \( \text{take}(\sigma_1, \sigma_2) = [2] \sqcup [2] = [2,2] \) and thus, \( \text{read}(\sigma_1, \sigma_2) \sqcup \text{take}(\sigma_1, \sigma_2) = [1,2,2,2] \) which is obviously not a multiset of \( M \). This is
in line with the intuitive meaning of independence because applying \( \sigma_1 \) to \( M \) results in the multiset \([1, 2]\) and does not allow application of \( \sigma_2 \). Nevertheless, applying \( \sigma_2 \) to \( M \) results in the same multiset \([1, 2]\), that allows the application of \( \sigma_1 \).

The following two lemmas state that for two independent substitutions, they can be applied in any order (and simultaneously) to the multiset, and the results are the same:

**Lemma 2.4** For two arbitrary substitutions \( \sigma_1 \) and \( \sigma_2 \) and a multiset \( M \), if \( M \models \sigma_1 \succ \sigma_2 \):

1. \( \text{read}(\sigma_1) \uplus \text{take}(\sigma_1) \subseteq M \) and \( \text{read}(\sigma_2) \uplus \text{take}(\sigma_2) \subseteq M \).
2. \( \text{read}(\sigma_1) \uplus \text{take}(\sigma_1) \subseteq M[\sigma_2] \) and \( \text{read}(\sigma_2) \uplus \text{take}(\sigma_2) \subseteq M[\sigma_1] \).

**Proof.** Proofs of all relations are trivial using the definition of independence. For sake of completeness, we give the proof for \( \text{read}(\sigma_2) \uplus \text{take}(\sigma_2) \subseteq M[\sigma_1] \), using the first relation in the lemma saying that \( \text{read}(\sigma_1) \uplus \text{take}(\sigma_1) \subseteq M \) and thus, \( \text{take}(\sigma_1) \subseteq M \):

\[
M \models \sigma_1 \succ \sigma_2 \Rightarrow \text{read}(\sigma_1, \sigma_2) \uplus \text{take}(\sigma_1, \sigma_2) \subseteq M \\
\Rightarrow (\text{read}(\sigma_1) \uplus \text{read}(\sigma_2)) \uplus (\text{take}(\sigma_1) \uplus \text{take}(\sigma_2)) \subseteq M
\]

Then, for an arbitrary element \( e \in U \):

\[
\text{max} (\text{read}(\sigma_1)(e), \text{read}(\sigma_2)(e)) + (\text{take}(\sigma_1)(e) + \text{take}(\sigma_2)(e)) \leq M(e) \\
\Rightarrow \text{read}(\sigma_2)(e) + \text{take}(\sigma_2)(e) \leq M(e) - \text{take}(\sigma_1)(e) \\
\Rightarrow \text{read}(\sigma_2)(e) + \text{take}(\sigma_2)(e) \leq M(e) - \text{take}(\sigma_1)(e) + \text{put}(\sigma_1)(e) \\
\Rightarrow \text{read}(\sigma_2) \uplus \text{take}(\sigma_2) \subseteq M[\sigma_1].
\]

**Proposition 2.5** For two arbitrary substitutions \( \sigma_1 \) and \( \sigma_2 \) independent with respect to \( M \):

1. \( (M[\sigma_1] \cdot \sigma_2) = (M[\sigma_2] \cdot \sigma_1) \).
2. \( M[\sigma_1, \sigma_2] = M[\sigma_2, \sigma_1] \).
3. \( (M[\sigma_1] \cdot \sigma_2, \sigma_2) = M[\sigma_2, \sigma_1] \).

**Proof.** Since \( M \models \sigma_1 \succ \sigma_2 \), due to Lemma 2.4 the max operators (in the definition of the \( \uplus \) operator) can be removed and subsequently added in the following. Thus, for an arbitrary element \( e \in U \):

\[
\]
The following two lemmas prove that if a substitution is independent from a multiple substitution, then the substitution and the substitutions the multiple substitution is composed from are pairwise independent and after application of one of the substitutions the others remain pairwise independent.

**Proposition 2.6** For all multisets \( M \) and substitutions \( \sigma_1, \sigma_2, \sigma_3 \):

\[
M \models \sigma_1 \bowtie (\sigma_2, \sigma_3) \Rightarrow \quad M[\sigma_1] \models \sigma_2 \bowtie \sigma_3
\]

\[
\wedge \quad M[\sigma_2] \models \sigma_1 \bowtie \sigma_3
\]

\[
\wedge \quad M[\sigma_3] \models \sigma_1 \bowtie \sigma_2
\]

---

1. \([M[\sigma_1]][\sigma_2](e) = ((M[\sigma_1] \boxslash \text{take}(\sigma_2)) \boxslash \text{put}(\sigma_2))(e)\)
2. \([M[\sigma_1, \sigma_2]](e) = ((M \boxslash \text{take}(\sigma_1, \sigma_2)) \boxslash \text{put}(\sigma_1, \sigma_2))(e)\)
3. \([M[\sigma_1, \sigma_2]](e) = ((M \boxslash \text{take}(\sigma_1, \sigma_2)) \boxslash \text{put}(\sigma_1, \sigma_2))(e)\)
Proof. We prove only the first conjunct. The rest can be proved following the same reasoning:

\[
M \models \sigma_1 \bowtie (\sigma_2, \sigma_3) \\
\Rightarrow \quad read(\sigma_1, (\sigma_2, \sigma_3)) \uplus take(\sigma_1, (\sigma_2, \sigma_3)) \subseteq M
\]

For an arbitrary element \( e \in U \):

\[
\begin{align*}
&\quad read(\sigma_1, (\sigma_2, \sigma_3))(e) + take(\sigma_1, (\sigma_2, \sigma_3))(e) \\ 
&\quad \Rightarrow \quad \max(read(\sigma_1)(e), \max(read(\sigma_2)(e), read(\sigma_3)(e))) \\ 
&\quad + take(\sigma_1)(e) + take(\sigma_2)(e) + take(\sigma_3)(e) \leq M(e) \\ 
&\quad \Rightarrow \quad \max(read(\sigma_2)(e), read(\sigma_3)(e)) \\ 
&\quad + take(\sigma_2)(e) + take(\sigma_3)(e) \leq M(e) - take(\sigma_1)(e) \\ 
&\quad \Rightarrow \quad \max(read(\sigma_2)(e), read(\sigma_3)(e)) \\ 
&\quad + take(\sigma_2)(e) + take(\sigma_3)(e) \leq M(e) - take(\sigma_1)(e) + put(\sigma_1)(e) \\ 
&\quad \Rightarrow \quad M[\sigma_1] \models \sigma_2 \bowtie \sigma_3.
\end{align*}
\]

\( \Box \)

**Proposition 2.7** For all multisets \( M \) and substitutions \( \sigma_1, \sigma_2, \sigma_3 \):

\[
M \models \sigma_1 \bowtie (\sigma_2, \sigma_3) \Rightarrow M \models \sigma_1 \bowtie \sigma_2 \land M \models \sigma_1 \bowtie \sigma_3
\]

**Proof.** Similar to the previous proposition. \( \Box \)

**Definition 2.8 (Constituents of a substitution)** The constituents of a substitution is the multiset of single substitutions present in a (single or multiple) substitution \( \sigma \), denoted by \( Const(\sigma) \), and defined inductively as follows:

1. for an arbitrary single substitution \( \sigma_1 \): \( Const(\sigma_1) = [\sigma_1] \)

2. for an arbitrary multiple substitution \( \sigma_1, \sigma_2 \): \( Const(\sigma_1, \sigma_2) = Const(\sigma_1) \uplus Const(\sigma_2) \).

The notion of constituents of a substitution is defined to formalize one of the main properties that we sought from our independence relation. Namely, we want the independence relation between two substitutions \( \sigma_1 \) and \( \sigma_2 \) to hold with respect to a multiset \( M \) if and only if the constituents of the two substitutions \( \sigma_1 \) and \( \sigma_2 \) can be applied in any order to \( M \). The proof needs the following lemma.

**Lemma 2.8** For two arbitrary substitutions \( \sigma_1 \) and \( \sigma_2 \), and a multiset \( M \), \( M \models \sigma_1 \bowtie \sigma_2 \) if and only if for all substitutions \( \sigma_0, \sigma_1, \ldots, \sigma_n \) in the constituents of multiple substitution \( (\sigma_1, \sigma_2) \), we have \( \bigcup_{i=0}^{n} read(\sigma_i') \uplus \sum_{i=0}^{n} take(\sigma_i') \subseteq M \), where \( \bigcup \) and \( \sum \) are generalizations of the \( \cup \) and \( \oplus \) operators.
Proof. \( M \models \sigma_1 \Rightarrow \sigma_2 \iff \text{take}(\sigma_1, \sigma_2) \sqcup \text{read}(\sigma_1, \sigma_2) \sqsubseteq M \iff (\text{by unfolding the definition of take}(\sigma_1, \sigma_2) \text{ and read}(\sigma_1, \sigma_2)) \left( \bigcup_{i=0}^{n} \text{read}(\sigma_i') \right) \sqcup \sum_{i=0}^{n} \text{take}(\sigma_i') \sqsubseteq M \) for all \( \sigma_i' \) \((0 \leq i \leq n)\) substitutions in the constituents of \( \sigma_1, \sigma_2 \).

\( \Box \)

**Theorem 2.9** Two substitutions \( \sigma_1 \) and \( \sigma_2 \) are independent with respect to a multiset \( M \), if and only if for the constituents of \( \sigma_1, \sigma_2 \) of the form \([\sigma_0', \sigma_1', \ldots, \sigma_n']\), for all \( i \) with \( 0 \leq i \leq n \), \( \sigma_i' \Rightarrow \sigma_0', \ldots, \sigma_{i-1}', \sigma_{i+1}', \ldots, \sigma_n' \).

**Proof.** If \( M \models \sigma_1 \Rightarrow \sigma_2 \), according to Lemma 2.8, \( \left( \bigcup_{i=0}^{n} \text{read}(\sigma_i') \right) \sqcup \sum_{i=0}^{n} \text{take}(\sigma_i') \sqsubseteq M \) but since multiset summation and union are commutative and associative, \( (\text{read}(\sigma_0') \cup (\text{read}(\sigma_0') \cup \ldots \cup (\text{read}(\sigma_{i-1}') \cup \text{read}(\sigma_{i+1}') \cup \ldots \cup \text{read}(\sigma_n')))) \sqcup (\text{take}(\sigma_i') \sqcup \ldots \sqcup \text{take}(\sigma_i') \cup (\text{take}(\sigma_i') \sqcup \ldots \sqcup \text{take}(\sigma_i') \sqcup \text{take}(\sigma_i'))) \subseteq M \), folding the union and summation expressions results in \( \text{read}(\sigma_0') \cup \text{read}(\sigma_0', \ldots, \sigma_{i-1}', \sigma_{i+1}', \ldots, \sigma_n') \sqcup (\text{take}(\sigma_i') \sqcup (\text{take}(\sigma_0', \ldots, \sigma_{i-1}', \sigma_{i+1}', \ldots, \sigma_n')) ) \subseteq M \). It follows from the definition of independence that \( M \models \sigma_i' \Rightarrow (\sigma_0', \ldots, \sigma_{i-1}', \sigma_{i+1}', \ldots, \sigma_n') \) (The reasoning holds in the other direction, too).

\( \Box \)

The above propositions and the last theorem show that the formal definition of independence matches the informal intuition behind the definition of independence which is to be used to model parallel tasks in GAMMA semantics.

In [8], simpler notions of substitution and independence are defined. There, a substitution \( \sigma_1 = N_1/N_2 \) is applied to the multiset \( M \) by just removing \( N_2 \) and adding \( N_1 \) if \( N_2 \sqsubseteq M \). Composition of two single substitutions \( \sigma_1 = N_1/N_2 \) and \( \sigma_2 = N_2/N_3 \) is defined as another single substitution \( N_1 \sqcup N_2/N_3 \sqsubseteq M \). Subsequently, the independence condition for the above two substitutions is defined to hold if and only if \( N_1 \sqsubseteq M[\sigma_2] \land N_2 \sqsubseteq M[\sigma_1] \). Although these definitions are simpler than the ones we gave before, they are not adequate for our purpose and lack properties like those of Proposition 2.5. As a counterexample, recall the substitutions of Example 2.1. Substitutions \( \sigma_1 = [2]/[1, 2] \) and \( \sigma_2 = [2]/[2, 2] \) are independent (according to both definitions) with respect to \( M = [1, 2, 2] \). However, their composition according to [8] \( (2 \sqcup 2)/(1, 2 \sqcup 2, 2) \) \( [2, 2] \) is not applicable to \( M \), thus violating Proposition 2.5. In other words, the composition of two independent substitutions as defined in [8] may grow larger than its intuitive meaning and thus it may cause both problems in the semantics of GAMMA and prevent some logically parallel execution from being executed simultaneously.

Our definition of independence is close to the definition of detached bindings in [11]. However, there, the author does not use the notion of independence to define the composition of independent tasks since he aims at an interleaving semantics.

14
2.3 GAMMA

We defined the notion of a shared multiset and basic operations on it and proved some properties. In this subsection, first, the structure of multiset elements and basic terms are defined. Then, we define the syntax and semantic of the GAMMA based on previous definitions.

2.3.1 GAMMA Shared Multiset

So far, we have not mentioned anything about the elements of the universal set. In the remainder of our discussion, the universal set is constructed from a base set $B$ containing atomic elements. The base set is associated with a set of relations and functions forming a logical structure $\mathcal{B} = (B, r_1, \ldots, r_n, f_1, \ldots, f_m)$, where $r_i \subseteq B^j$ is a relation with fixed arity $j_i$, and $f_i : B^{k_i} \to B$ is a function with fixed arity $k_i$. Later on, we base the syntax of our conditional and multiset expressions on this structure and, hence, dispense with re-stating them in our language definition. A more complete text on logical structures can be found in [28].

**Definition 2.9 (Term)** The set of terms from a logical structure $\mathcal{B}$ (containing constant set $B$) and a set of variables $V$ (disjoint from $B$), denoted by $T(\mathcal{B}, V)$, is the minimal set satisfying the following properties:

1. for all $b \in B$, $b \in T(\mathcal{B}, V)$;
2. for all $v \in V$, $v \in T(\mathcal{B}, V)$;
3. for all $f_i$ in $B$ with arity $k_i$ and for all $t_1, \ldots, t_k \in T(\mathcal{B}, V)$, $f_i(t_1, \ldots, t_k) \in T(\mathcal{B}, V)$.

The set of closed terms $CT(\mathcal{B}, V)$ includes only those elements of $T(\mathcal{B}, V)$ that do not contain any variable. Conditional expressions can be built the same way as the terms using relations $r_i$ among terms. In this report, we restrict our attention to the conditions expressed by propositional logic formulae.

To define the elements of the universal set, we close the terms under pairing and call it basic expression:

**Definition 2.10 (Basic Expression)** The set of basic expressions $BasicExp$, based on the set of terms $T(\mathcal{B}, V)$ is the minimal set satisfying the following properties:

1. for all $t \in T(\mathcal{B}, V)$, $t \in BasicExp$;
2. for all $b_1, b_2 \in BasicExpr$, $(b_1, b_2) \in BasicExp$.

In this report, the universal set $U$ is the set of basic expressions based on closed terms.
\[ Program ::= ProgramName = \{ Rules \} \]
\[ Rules ::= Rule | Rule, Rules \]
\[ Rule ::= RuleName = MultisetExp \rightarrow MultisetExp \Leftarrow Condition \]
\[ MultisetExp ::= \varepsilon | MExp \]
\[ MExp ::= BasicExp | BasicExp, MExp \]

Figure 2.3: Basic GAMMA Syntax

### 2.3.2 GAMMA Syntax

The syntax of basic GAMMA programs is presented in Figure 2.3. A GAMMA program consists of a name (a string of syntactic class \texttt{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 \texttt{RuleName}) and a set of terms in the left- and right-hand side of the substitution arrow \( \rightarrow \) and a condition part that are to be evaluated by the multiset content. In the given syntax, \texttt{BasicExp} and \texttt{Condition} non-terminals refer to what we have defined in Definition 2.9.

Either of the multiset expressions in the right or left side of a rule can be empty (represented by the empty string \( \varepsilon \)). We use the rule with both sides empty and condition \texttt{true} as the notion for the \texttt{idle} rule which is always enabled and does not change the multiset.

**Example 2.2 (A rule for addition)** The following is a simple example of a GAMMA rule taking two values from a multiset, and replacing them by their sum:

\[ Add = x, y \rightarrow x + y \Leftarrow true \]

The condition is assumed to be \texttt{true} when left implicit throughout the rest of the paper.

**Example 2.3 (Sorting)** A typical example of a GAMMA program is sorting in ascending order:

\[ Sort = \{ Swap = (i, x), (j, y) \rightarrow (i, y), (j, x) \Leftarrow (i < j) \wedge (y < x) \} \]

The above program repeatedly substitutes two members (denoted by a \texttt{position, value} pair) of a list or an array. The \texttt{Swap} rule is applied only if the relative position of two elements is not in line with ascending order.

To make the basic GAMMA theory more usable, we add some syntactic sugar for strengthening rule conditions and representing the rules that cannot perform any computation:
Definition 2.11 (Strengthening condition) The strengthening of a rule \( r = \text{LEexp} \rightarrow \text{REexp} \Leftarrow \text{Cond} \) by a condition \( \text{Cond}_2 \) is defined as: \( \text{Cond}_2 \triangleright r \triangleq \text{LEexp} \rightarrow \text{REexp} \Leftarrow \text{Cond}_1 \land \text{Cond}_2 \).

Definition 2.12 (Skip rule) We define the rule \( \text{skip} \triangleq e \mapsto e \Leftarrow \text{false} \).

In other words, \text{skip} represents a rule that can never be enabled.

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 manipulation 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 write and left hand side of the rule. To represent this notion more concisely, we define the following syntactic shorthand:

Definition 2.13 (Explicit read) When an expression \( e \) is only read from the multiset and no manipulation on it is required, then we denote this by \( e? : \text{LEexp} \rightarrow \text{REexp} \Leftarrow \text{Cond} \), which is defined to be \( e, \text{LEpr} \mapsto e, \text{REexp} \Leftarrow \text{Cond} \). Similarly, a multiset expression (a list of basic expressions) can be read using ? notion after each basic term.

2.3.3 GAMMA Semantics

Execution of a GAMMA rule deals with manipulation of the shared multiset by reading and taking data objects from the multiset and putting back the results. We have defined the notion of substitution previously in order to represent execution and computation in our semantics. A rule terminates when there are not enough enabling elements in the multisets. The execution and termination of a program depends on the execution and termination of its single rules. Note that our notion of rule termination is different from the usual process algebraic notion of termination in that a terminated rule in context of a program may become enabled later due to execution of other rules.

We aim to prevent imposing behavioral and structuring constraints (such as ordering and sequentiality) on rule execution at this level of abstraction. Hence, we allow arbitrary order of rule execution and arbitrary level of parallelism between them in the GAMMA program semantics. So, the first step in defining GAMMA program semantics is to define the semantics of a single rule execution / termination; the second step is composing execution of program rules in any possible (chaotic) ordering.

To define the semantics of a single GAMMA rule, we assume all variables in an expression (in a rule) to range over the base set \( B \). This simplifies the discussion by preventing type information and type checking problems. However, for a complex system, using a single base set for all variables may entangle the specification of rules, and make them less readable. We admit this problem and we propose further extensions to deal with it in the final section.
Definition 2.14 (Enabling valuation) A valuation function $\overline{v} : V \rightarrow B$ enables a rule $r : \text{Name} = L\text{Expr} \rightarrow R\text{Expr} \iff \text{Cond}$ for a multiset $M$, denoted by $M, \overline{v} \models r$, if and only if:

1. The valuation causes the left-hand side multiset expression of $r$ to be a multiset subset of $M$: $\overline{v}[L\text{Expr}] \subseteq M$ (where $\overline{v}[\text{expr}]$ is the multiset resulting from substitution of all occurrences of each variable $v \in V$ with its corresponding value $\overline{v}(v)$ in expression $\text{expr}$).

2. It satisfies the condition part of $r$: $B, \overline{v} \models \text{Cond}$.

Note that a more precise definition of $\overline{v}[\text{expr}]$ would need lifting the valuation function to terms and lifting the new function one more time to the list of terms. The possible existence of several enabling valuations in multiset content for a single rule, provides another notion of abstraction in GAMMA. For example, as shown in the semantics given below, a single rule may cause different computations and even these computations may be composed in parallel or in sequence.

The GAMMA semantics is presented in two parts in the style of Structured Operational Semantics (SOS) [23]. The first part, presented in Figure 2.4, shows when a single computation is performed (denoted by $\rightarrow_1$, where $\sigma$ is the corresponding single substitution) or when a single rule terminates (denoted by $\sqrt{1}$). The second part, Figure 2.5, shows how computations / termination of a program (set of rules, denoted by $\vdash$ and $\checkmark$, respectively) are composed of computation / termination of single rules.

We separated the two parts of the semantics in order to re-use the first part in both defining the chaotic behavior of GAMMA programs (the second part) and also coordinated behavior of schedules (the next section). 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 functionality with an operational semantics of another functionality model (say, Java methods, or even a hierarchy of interface services), and benefit from the specification and verification models presented in the remainder. In such cases, enough care should be taken in order not to loose the orthogonality in the model.
(ProgComp) \( r \in R \quad (r, M) \xrightarrow{\sigma_1} (r, M_1) \)
\[ (R, M) \xrightarrow{\sigma_2} (R, M_1) \]
(R, M) \xrightarrow{\sigma_1 \lor \sigma_2} (R, M[\sigma_1, \sigma_2])

(ProgPar) \( \forall r \in R \quad (r, M) \xrightarrow{1} (R, M_1) \)
\[ (R, M) \xrightarrow{2} (R, M_2) \]
\( \forall r \in R \quad (r, M) \xrightarrow{\sqrt{1}} \)

Figure 2.5: GAMMA Semantics: Chaotic Execution

In the given semantics, the tuples \((r, M)\) and \((R, M)\) represent the state of a system as the GAMMA rule or program \((r \text{ or } R)\) and the multiset content \((M)\). Since GAMMA rules remain unchanged during each step in the operational semantics we could have omitted them from the state definition and added them to the transition part (as a subscript \(\xrightarrow{\sigma} R\), for example). However, we followed the current style for readability and also for representation consistency with the semantics of schedules (presented in the next section).

In the semantics of the first part (Figure 2.4), the rule \((\text{RuleTerm})\) specifies that a rule \(r\) terminates with respect to a multiset \(M\) if there is no enabling valuation for the rule in the multiset. The rule \((\text{RuleComp})\), expresses that a single computation is performed (by replacing the valuation of the left-hand side with the right-hand side in the multiset content) if there exists an enabling valuation from the multiset. The semantics of the basic functionality model is the smallest transition relation satisfying the rules given in Figure 2.4.

We continue with defining semantics of GAMMA programs. Figure 2.5 presents chaotic execution of GAMMA programs by using the semantics of a single rule execution and termination. The rule \((\text{ProgComp})\) uses the semantics of single rule execution and extends it to program computation. It specifies that if a single rule can perform a computation, any program containing such a rule can do so. The rule \((\text{ProgPar})\) shows the possible parallelism in the execution of (a set of) rules. It specifies that if a program can perform two independent computations with respect to a multiset, then it can perform both computations concurrently. Obviously, repeated application of this rule allows any possible level of parallelism in the execution of a GAMMA program. The rule \((\text{ProgTerm})\) specifies termination of a GAMMA program. It specifies that termination of a program can be deduced from termination of all its rules.

The semantics for GAMMA programs is taken from [8] (with presentation changes and the modifications concerning the independence relation discussed before). The rule for parallelism is different from [6] in that concurrency is modelled with a step semantics instead of an interleaving one. This decision in selecting a step semantics for GAMMA is made to be able to distinguish between really parallel executions and permutations of sequential executions.
in the coordination model (and have both notions available in the semantics), which is presented in the remainder.

**Example 2.4 (Sorting)** Execution of the sorting program (presented in Example 2.3) on the initial multiset $M = [(0, 5), (1, 3), (2, 7), (3, 4)]$ results in a transition system depicted partially in Figure 2.6. To provide a more compact picture of the transition system, instead of presenting the multiset content in each state (set of pairs), the resulting list or array is shown as a tuple. Also, we omitted the substitution labels on transitions for readability. However, in the semantics we need to keep the information about substitution on transitions to specify parallel execution of rules.

In this example, a single rule *Swap* is applied to the multiset, but due to different enabling valuations for this rule, it causes several different substitutions. Among the substitutions, those swapping two disjoint pairs of values are independent and can be done in parallel. For example, the transition from $(5, 3, 7, 4)$ to $(3, 5, 4, 7)$ is the result of parallel execution of two substitutions: one swapping the relative position of 3 and 5 and the other of 4 and 7. The dashed line shows the transition resulting from rule *(ProgPar)*.

**Example 2.5 (Dining philosophers)** A number of $n$ philosophers can sit around a table with a big bowl of spaghetti in the center, $n$ china dishes, and $n$ forks in between each pair of philosophers. Philosophers' lives consist of thinking, getting hungry and as a result eating spaghetti. But, the problem lies in the complex nature of spaghetti which requires each philosopher to claim both adjacent forks to eat.

Each philosopher, after an arbitrary amount of time spending on philosophical thoughts, gets hungry, sits down on his/her chair and tries to get hold of both forks on the right and the left. Unfortunately, one (or both) of these forks
may already be in use by colleagues and s/he should wait for them to finish eating.

The informal definition of the dining philosophers problem is formalized in terms of GAMMA rules (The given rules are slightly different versions of the ones in [6], also adapted to match our syntax):

\[
\text{DiningPhilosophers} = \begin{cases}
\text{Sit} = (P, i) \rightarrow (PS, i), \\
\text{Eat} = (PS, i), (F, i), (F, (i + 1) \mod n) \rightarrow (PE, i), \\
\text{Leave} = (PE, i) \rightarrow (P, i), (F, i), (F, (i + 1) \mod n)
\end{cases}
\]

In the above GAMMA program, thinking philosophers are denoted by \((P, i)\), forks beside them by \((F, i)\) and \((F, (i + 1) \mod n)\), hungry (sitting) philosophers by \((PS, i)\), and eating philosophers by \((PE, i)\). The system consists of three rules for modelling actions representing philosophers getting hungry and sitting around the table (Sit), claiming forks and eating (Eat), and returning to normal (thinking) condition (Leave). The initial multiset of the system (room) is specified by: room = \{\((P, 0), (F, 0), \ldots, (F, n - 1), (F, n - 1)\)\].

Note that chaotic execution of the GAMMA program satisfies the definition of the problem and prevents two adjacent philosophers to get hold of one fork at the same time (mutual exclusion). However, the chaotic execution still does have the problem of individual starvation (i.e., a philosopher may remain seated forever without being fed). We return to the properties of this program at the end of this chapter.

It is worth mentioning that a GAMMA program, if designed in an appropriate way, is often abstract from the multiset size and thus scalable to different problem sizes without any change. In the above example, adding a new philosopher to the problem does not need any change to the program (assuming that \(n\) works like a parameter of the program). Although this notion of abstraction seems primitive in this problem, it could be quite useful in more complicated designs.

**Example 2.6 (An elevator system)** In this example, an elevator system is being specified. Our elevator system consists of an elevator moving up and down between floors of a building (numbered from 0 to MaxFloor). At each floor, there is a push button to announce a request for the elevator when turned on. By arrival of the elevator on a floor and servicing the requests, the request flag is turned off automatically. The same setting works for the push buttons inside the elevator, used to indicate the requested stops for passengers inside.

To model such a system, we propose a multiset containing events of requests for elevator stops represented by \(((\text{inStop}, i), \text{status})\) and \(((\text{exitStop}, i), \text{status})\) that show the status of the request button for the \(i\)’th floor, inside and outside the elevator, respectively (status can have the value on or off). Also, to indicate the status of the door, the tuple \((\text{door}, \text{status})\) is placed in the multiset (with the status being closed or open). The tuple \((cf, i)\) shows at which floor the elevator currently resides.
A GAMMA program for the elevator system is as follows:

\[
\begin{align*}
\text{ElevatorSystem} = \\
\{ \\
\quad \text{inRequest} = ((\text{inStop},i), \text{off}) \rightarrow ((\text{inStop},i), \text{on}), \\
\quad \text{extRequest} = ((\text{extStop},i), \text{off}) \rightarrow ((\text{extStop},i), \text{on}), \\
\quad \text{moveUp} = (\text{door}, \text{closed})? : (\text{cf},i) \rightarrow (\text{cf},i+1) \Leftarrow i < \text{MaxFloor}, \\
\quad \text{moveDown} = (\text{door}, \text{closed})? : (\text{cf},i) \rightarrow (\text{cf},i-1) \Leftarrow i > 0, \\
\quad \text{close} = (\text{door}, \text{open}) \rightarrow (\text{door}, \text{closed}), \\
\quad \text{open} = (\text{door}, \text{closed}) \rightarrow (\text{door}, \text{open}), \\
\quad \text{load} = (\text{door}, \text{open})? : (\text{extStop},i), \text{on} \rightarrow ((\text{extStop},i), \text{off}), \\
\quad \text{unload} = (\text{door}, \text{open})? : (\text{inStop},i), \text{on} \rightarrow ((\text{inStop},i), \text{off}) \\
\}\end{align*}
\]

The initial multiset for this system is defined as:

\[
\text{State} = [((\text{inStop},0), \text{off}), \ldots, ((\text{inStop},\text{MaxFloor}), \text{off}), \\
((\text{extStop},0), \text{off}), \ldots, ((\text{extStop},\text{MaxFloor}), \text{off}), \\
(\text{door}, \text{open}), (\text{cf},0)]
\]

which shows that there is no request for the elevator that is staying at the ground floor with the door open.

The chaotic execution of the GAMMA program for the elevator example allows the elevator to go up and down the floors without servicing any request. Following [8], to solve problems of this nature, we use a coordination language for governing the GAMMA chaotic executions, as explained in the next section.

In the remainder of this section, we formalize some properties of the semantics of GAMMA programs. The following lemma and corollary state that if a program can perform a computation, then a larger program can do the same, too:

**Lemma 2.10** If a program \( R \) can perform a \( \sigma \) transition, i.e., \( (R, M) \xrightarrow{\sigma} (R, M') \), then for an arbitrary rule \( r \), \( (R \cup \{r\}, M) \xrightarrow{\sigma} (R \cup \{r\}, M') \).

**Proof.** By an induction on the proof length for \( (R, M) \xrightarrow{\sigma} (R, M') \). See Appendix A for the detailed proof.

\[ \square \]

**Corollary 2.11** For GAMMA programs \( R_1 \) and \( R_2 \), if \( (R_1, M) \xrightarrow{\sigma} (R_1, M') \), then \( (R_1 \cup R_2, M) \xrightarrow{\sigma} (R_1 \cup R_2, M') \).

The following lemma defines how computations of a program can be decomposed to (or composed from) computations of an arbitrary rule in the program and the rest of the program without this rule. This lemma is particularly helpful in reasoning about properties of program computations inductively (by reducing the size of the program in the induction):
Lemma 2.12 For an arbitrary GAMMA program $R$, multisets $M$ and $M'$, and substitution $\sigma$, $(R, M) \rightarrow (R, M[\sigma])$ if and only if for all rules $r \in R$, one of the following cases holds:

1. $(R \setminus \{r\}, M) \rightarrow (R \setminus \{r\}, M[\sigma])$

2. $(R \setminus \{r\}, M) \rightarrow (R \setminus \{r\}, M[\sigma_1]), (\{r\}, M) \rightarrow (\{r\}, M[\sigma_2]), \sigma = \sigma_1, \sigma_2$ and $M \models \sigma_1 \Rightarrow \sigma_2$.

3. $(\{r\}, M) \rightarrow (\{r\}, M[\sigma])$

Proof. See Appendix A.

2.4 Coordination

The goal of our coordination language is to specify restricted behavior of GAMMA programs. Hence, it should provide composition operators to structure execution and restrict the chaotic behavior of GAMMA rules.

2.4.1 Coordination Language Syntax

The syntax of a coordination language is specified in Figure 2.7. This syntax is a slightly modified version of the one given in [8].

In this syntax, Rule.Name is the notation to adopt GAMMA rules as the building blocks of Schedules (the coordination-language expressions). The rule-conditional operator $\checkmark$ is used to provide different strategies based on whether or not a rule can be executed. If the rule in the left operand can be executed, then the first schedule at its right-hand side is chosen for execution. Otherwise, the second schedule is chosen for execution. Note that, unlike [8], we only check the possibility of rule execution in the rule conditional operator (in [8] denoted by $\rightarrow$) and do not necessarily execute it. This becomes clearer in the semantics given below. As we show in the remainder, this change is useful in defining a simpler notion of general schedules that mimic the behavior of GAMMA program executions. Nevertheless, the $\rightarrow$ operator can be defined syntactically using our rule conditional operator semantics, as shown below.

The sequential and parallel operators (; and $|$), respectively) are also included in the coordination language to provide appropriate schedule compositions. The recursion operator $\mu$ is used to make recursive schedules ($\mu z. s(x)$) explicitly. Only schedules in which all recursion variables are bound by $\mu$ are of interest in this report. In the rest of this report, we usually define a name for schedules. These names serve as a syntactic shorthand for the defined schedules. To use definitions of other schedules in a new schedule composition (in a hierarchical structure), one can refer to their name.
2.4.2 Coordination Language Semantics

The SOS rules for the semantics of the coordination language are given in Figure 2.8. In this semantics, the rules (Term) and (Comp) show how termination and computation of a rule relate to computation and termination of its corresponding schedules. The rules (RC0) to (RC3) represent the semantics of the rule-conditional operator. The rule (RC0) specifies that if the rule \( r \) can be scheduled then the first argument at the right hand side of the rule conditional is selected for execution; otherwise, according to (RC1), the second schedule is started. The rules (RC2) and (RC3) specify that when the selected argument (according to the rule condition) terminates, then the rule conditional terminates.

Obviously, the semantics of the above rules depends on the semantics of a single rule given in the GAMMA semantics. This fact is denoted in the semantics by the use of single-rule transition \( \rightarrow_t \) and termination \( \checkmark \). The rule-conditional-execution operator \( \rightarrow \) of \([8]\) is defined using our rule-conditional operator as follows:

\[ r \rightarrow s[t] \overset{\Delta}{=} r \land (r ; s)[t] \]

We use \( r \rightarrow s \) and \( r \land s \) as shorthands for \( r \rightarrow s[\text{skip}] \) and \( r \land s[\text{skip}] \).

The rules (S0), (S1), and (S2) are dedicated to behavior and termination of the sequential composition operator. Note that, we need rules like (RC1) and (S1), because we only allow action transitions and for example, unlike \([8]\), we do not allow empty transitions. The rules (P0) to (P3) provide the semantics for the parallel-composition operator.

The rules (R0) and (R1) define recursion, where \( s[y/x] \) is the syntactic substitution of recursion variable \( x \) with expression \( y \). A recursive schedule can perform a computation or terminate if the corresponding schedule after removing the recursion operator and substituting the recursion variable by the whole (recursive) schedule can perform that computation or can terminate.

The correctness of the semantics highly relies on the properties of multiset substitution that we proved in Section 2.2. For example, using Theorem 2.9 and an induction on the depth of the proof for transitions, we are able to prove the following propositions:

---

Schedule ::= RuleName

| RuleName \& Schedule
| Schedule ; Schedule
| Schedule \| Schedule
| \( \mu \text{RecursionVar} \). Schedule
| RecursionVar

Figure 2.7: Coordination Language Syntax
Figure 2.8: Coordination Language Semantics
Proposition 2.13 For arbitrary schedules \( s \) and \( s' \), arbitrary multisets \( M \) and \( M' \), and all substitutions \( \sigma \), if \( (s, M) \xrightarrow{\sigma} (s', M') \) is a valid computation according to the semantics, then \( \text{read}(\sigma) \supseteq \text{take}(\sigma) \subseteq M \) and \( M' = M[\sigma] \).

Proof. By an induction on a proof depth for the transition \( \sigma \):

1. Induction base: Transitions with the proof of depth 1 can be only due to (Comp) (since other rules call for a proof of a non-trivial transition in their premises, they need a proof of depth, at least, 2). Thus, \( s \equiv r \) and \( s' \equiv \text{skip} \), where \( r \) is a GAMMA rule, and \( (r, M) \xrightarrow{\sigma} (\text{skip}, M') \). But according to the GAMMA single rule semantics (enabling valuation definition), if \( \sigma \) is of the form \( N/N' \), then \( N' \subseteq M \). Hence, \( \text{read}(\sigma) \supseteq \text{take}(\sigma) \subseteq M \) and (according to the semantics) \( M' = M[\sigma] \).

2. Induction step: We have to prove that if \( (s, M) \xrightarrow{\sigma} (s', M') \) then \( \text{read}(\sigma) \supseteq \text{take}(\sigma) \subseteq M \) and \( M' = M[\sigma] \). This statement follows trivially from analyzing the transition rules in the coordination semantics and using induction hypothesis on the premises.

Proposition 2.14 For all schedules \( s \) and \( s' \), all multisets \( M \) and \( M' \), and all substitutions \( \sigma_1 \) and \( \sigma_2 \), if \( (s, M) \xrightarrow{\sigma_1 \circ \sigma_2} (s', M') \), then \( M \models \sigma_1 \leftrightarrow \sigma_2 \) (Thus, constituents of \( \sigma_1 \) and \( \sigma_2 \) can be applied in any order to \( M \)).

Proof. Similar to the proof of the previous proposition.

Example 2.7 (Dining philosophers. Coordination) In this example, three different coordination expressions are presented for the rules given in Example 2.5. First, we give a schedule that executes rules in their original chaotic manner:

\[
\text{ServeChaotic} = ((\mu X. \text{Sit} \land (\text{Sit} \parallel X)) \parallel (\mu X. \text{Eat} \land (\text{Eat} \parallel X)) \parallel (\mu X. \text{Leave} \land (\text{Leave} \parallel X))
\]

This schedule replicates arbitrary copies of rules in parallel, and begins executing them. Since, the semantic of parallel composition allows execution of components in sequence (due to rules (P0) and (P1)) this schedule allows arbitrary parallelism and ordering among rules. We will formalize this intuitive notion of chaotic schedules in the remainder.

The second schedule is a sequential schedule that first lets all philosophers think and get hungry, then it begins serving them and letting them leave the
table one by one:

\[
ServeSeq = \mu X. Think ; ServeP_0 ; \ldots ; ServeP_{n-1} ; X
\]

\[
Think = \bigvee_{j=0}^{n-1} Sit
\]

\[
ServeP_j = Eat_j ; Leave_j
\]

(Eat \equiv (j = i) \triangleright Eat)

(Leave \equiv (j = i) \triangleright Leave)

(for 0 \leq j < n)

Note that schedules like \(ServeP_j\) are a shorthand for a set of schedules each representing the servicing of a particular philosopher. We use this shorthand in other places in the rest of the paper as well. Also, \(\bigvee_{j=0}^{n-1}\) is a shorthand for parallel composition of schedules. This shorthand notation needs commutativity and associativity of parallel composition which will be proved in the next section.

The third schedule allows philosophers to eat together. This is done by grouping odd numbered and even numbered philosophers and letting them eat together:

\[
ServePar = \mu X. (Think ; (Evens \parallel Odds)) ; X
\]

\[
Think = \bigvee_{j=0}^{n-1} Sit
\]

\[
Evens = \bigvee_{j=0}^{n-1} \bigvee_{j=2}^{n+2} ServeP_{2j}
\]

\[
Odds = \bigvee_{j=0}^{n+2} ServeP_{2j-1}
\]

\[
ServeP_j = Eat_j ; Leave_j
\]

(Eat \equiv (j = i) \triangleright Eat)

(Leave \equiv (j = i) \triangleright Leave)

(for 0 \leq j < n)

Example 2.8 (The elevator system, Coordination) As before, a general schedule would represent the chaotic behavior of the program. For sake of brevity, we dispense with presenting the same chaotic schedule for this example. The general form of such chaotically behaving schedule is given in Section 2.4.3.

An interesting schedule would be to control the elevator in such a way that it goes up from the ground floor to \(MaxFloor\), and then vice versa, stopping at each floor, and servicing any available request there. Requests from inside or outside can occur at any time spontaneously:

\[
ElevatorSchedule = ((\mu X.(inRequest \parallel X)) \parallel (\mu X.(extRequest \parallel X))) \parallel (\mu X.(ServiceUp ; ServiceDown) ; X)
\]

\[
ServiceUp = \mu X.(open ; (load \parallel unload) ; close ; (moveUp \rightarrow X))
\]

\[
ServiceDown = \mu X.(open ; (load \parallel unload) ; close ; (moveDown \rightarrow X))
\]

2.4.3 Equivalence of Schedules

To show the relationships between different schedules and also schedules and GAMMA programs we give the definition of bisimilar schedules (GAMMA programs) in this subsection. These definitions can be used as a basis for both refinement and process algebraic verification of schedules.
Definition 2.15 (Statebased Bisimulation Relation) We call a relation \( R \) on states a statebased bisimulation relation if and only if for any two states \( (s, M) \) and \( (t, M) \) such that \( (s, M) R (t, M) \), for all states \( (s', M_1') \) and \( (t', M_2') \):

1. \( (s, M) R (s', M_1') \) for some state \( (t'', M_1'') \), \( (t, M) R (t'', M_1'') \) for all \( M'' \), \( (s', M'') R (t'', M'') \)

2. \( (t, M) R (t', M_2') \) for some state \( (s'', M_2'') \), \( (s, M) R (s'', M_2'') \) for all \( M'' \), \( (s'', M'') R (t'', M'') \)

3. \( (s, M) \) \( R \) \( (t, M) \)

4. \( (t, M) \) \( R \) \( (s, M) \)

Two schedules \( s \) and \( t \) are bisimilar (written: \( s \equiv t \)) if and only if there exists a statebased bisimulation relation \( R \) such that for all multisets \( M \), \( (s, M) R (t, M) \).

Note that with being a little bit sloppy we use the notion of bisimulation relation and bisimilarity between two GAMMA programs (sets of GAMMA rules) and a GAMMA program and a schedule. In the definition of these three different bisimulations the transition arrows and termination ticks refer to transition and termination to the corresponding GAMMA or Coordination semantics.

Theorem 2.15 (Equivalence) Statebased bisimulation is an equivalence relation.

Proof. Straightforward from the definition of bisimulation.

Theorem 2.16 (Congruence) Bisimilarity is a congruence with respect to all coordination operators.

Proof. See Appendix A.

Proposition 2.11 Some bisimilar schedules: According to the given semantics, the following bisimilarities hold, for all schedules \( s, s_1, s_2, s_3 \) and rule \( r \):

1. \( \text{skip} ; s \equiv s \)
2. \( s ; \text{skip} \equiv s \)
3. \( s_1 ; (s_2 ; s_3) \equiv (s_1 ; s_2) ; s_3 \)
4. \( \text{skip} || s \equiv s \)
5. \( s_1 || s_2 \equiv s_2 || s_1 \)
6. \( (s_1 || s_2) || s_3 \equiv s_1 || (s_2 || s_3) \)
7. $r \triangleright \text{skip[skip]} \cong \text{skip}

8. $r \triangleright r[\text{skip}] \cong r

9. $(r \triangleright s_1[s_2]) ; s_2 \cong r \triangleright (s_1 ; s_2)[s_2 ; s_2]

10. If $y$ is not present in $s$, $\mu x.s \cong \mu y.s[y/x]$.

Proof. See Appendix A.

**Theorem 2.18 (Most General Schedule)** An arbitrary GAMMA program $R$, is bisimilar to the following schedule:

$$MGS(R) \triangleq \| \in R \ (\mu X.r \triangleright (r \parallel X)).$$

**Proof.** See Appendix A.

**Proposition 2.19** For two arbitrary GAMMA programs $R_1$ and $R_2$ and two arbitrary schedules $s$ and $t$, if $R_1 \cong s$ and $R_2 \cong t$, then $R_1 \cup R_2 \cong s \parallel t$.

**Proof.** See Appendix A.

### 2.4.4 Local Termination

In the given semantics for GAMMA programs, a program can terminate only when all rules agree on termination (i.e., when no rule finds an enabling valuation in the multiset). Similarly, the semantics of parallel composition in the schedules only allows the parallel composition to terminate, when both components terminate. In other words, given the current semantics, termination of parallel programs is a global issue for which all parallel components (rules, or schedules) have to coordinate. For reasons such as convenience in implementation, one may want to have a semantics where termination can be decided locally among the components. In such a semantics, components may decide to terminate and remove themselves from the system, when they do not find enabling valuations. To reflect such a decision, the following rules should be added to the semantics of GAMMA programs:

$$
\frac{(\text{ProgComp2})}{\{r\} \subset R \quad (r, M) \sqrt{1} \quad (R, M) \xrightarrow{\mathcal{S}_{R}} (R, M_1) \quad (R, M) \xrightarrow{\mathcal{S}_{R}} (R \setminus \{r\}, M_1)}
$$

Note that we require the removed set of rules to be a strict subset of the original program. This prevents introduction of empty programs. (Hence, we
use \( \{ r \} \subset R \) instead of \( r \in R \). Similarly, following rules should be added to the semantics of parallel composition:

\[
\begin{align*}
(P4) & \quad \frac{\langle s_1, M \rangle \vdash \langle s_2, M \rangle \xrightarrow{\alpha} \langle s_2', M_2 \rangle}{\langle s_1 \parallel s_2, M \rangle \xrightarrow{\alpha} \langle s_2', M_2 \rangle} \\
(P5) & \quad \frac{\langle s_1, M \rangle \xrightarrow{\alpha} \langle s_1', M_1 \rangle, \langle s_2, M \rangle \vdash \langle s_2', M_2 \rangle}{\langle s_1 \parallel s_2, M \rangle \xrightarrow{\alpha} \langle s_1', M_1 \rangle}
\end{align*}
\]

Since chaotic execution of programs and parallel execution of schedules do not impose any constraint about the order of execution on their components, the previous semantic rules (allowing global termination) should still remain in the semantics. So, the semantics with local termination only adds more option for termination in the execution of GAMMA programs and schedules. One reader can check that all previous proofs for equivalences (including that of the Most General Schedule) still hold with the new semantics.

It is essential to change both the semantics of GAMMA programs and parallel compositions (or only change the GAMMA program semantics); since a major feature of our approach is that adding a new aspect is only meant to restrict the behavior of the basic semantic model or to add information to it. However, changing the schedules without introducing the local termination in GAMMA program semantics adds some runs of execution to schedules (leading to termination) that are not possible in their corresponding GAMMA program execution. We decided to keep the current semantics of GAMMA programs (and thus semantics of schedules) to make our timed extensions a conservative extension of currently available GAMMA semantics.

### 2.5 Reasoning

The given semantics of GAMMA and the coordination language provides the basis for reasoning about chaotic and coordinated programs. Since we are aiming at using this method for the design of reactive (and later real-time) systems, it is important for us to be able to reason about properties of computation paths in the executions of a (coordinated) GAMMA program. Temporal Logics are useful means of expressing these kinds of properties [9]. Furthermore, widespread use of these types of logics provides a rich set of theoretic background, applications, and tools, which we can benefit from in this research.

Next, we define a valid computation trace of a coordinated program and use Linear-time Temporal Logic (LTL) as a specification language to reason about correctness of schedules. To do this, the notion of a valid run of a GAMMA schedule should be defined formally so that it can fit LTL’s linear frame.

**Definition 2.16 (State sequence)** A (possibly infinite) sequence \( SM_0. = \langle s_0, M_0 \rangle, (s_1, M_1), \ldots \) is a state sequence of the schedule \( s \) with initial multiset \( M \) if and only if it satisfies:
\[ \text{Formula ::= Atom | } \neg \text{Formula } | \text{Formula } \lor \text{Formula} \\
| X \text{Formula} | \text{Formula } U \text{Formula} \]

Figure 2.9: Syntax of Linear-time Temporal Logic Formulas

1. Initiality: \( M_0 = M \), \( s_0 = s \)

2. Sequence: for all \( i \geq 0 \), there exists a \( \sigma \) such that \( (s_i, M_i) \xrightarrow{\sigma} (s_{i+1}, M_{i+1}) \).

**Definition 2.17 (Run)** The sequence \( M_{0..} = M_0 M_1 .. \) is a valid run of schedule \( s \) and multiset \( M \) if and only if there exists a state sequence of \( s \) and \( M \), \( SM_{0..} = (s_0, M_0)(s_1, M_1) .. \).

The function \( |M_{0..}| \) is defined to return the number of multisets present in the sequence \( M_{0..} \), if it is finite, and \( \infty \) for infinite sequences. The formula \( i \leq \infty \) is assumed to be true for all \( i \in \mathbb{N} \).

Different syntaxes have been defined for LTL. The syntax we are going to use (presented in Figure 2.9) is taken from [9]. In this syntax, \( \text{Atom} \) means propositional logic formulas that can be evaluated using the multiset content in each state. The syntax of atomic propositions is the same as the syntax of conditions in GAMMA rules, except that the former cannot contain variables. Intuitively, \( X \varphi \) means that \( \varphi \) will hold in the next state, and \( \psi U \varphi \) means that \( \varphi \) will hold some time in a state in the finite future (or now), until which in all states \( \psi \) is true. The following useful temporal operators can be defined based on the above set of operators:

1. Eventually (finally): Some state in the future (or now) formula \( \varphi \) will hold:
   \[ F\varphi \overset{\Delta}{=} \text{true } U \varphi. \]

2. Globally (always): In all states (now and in the future), formula \( \varphi \) holds:
   \[ G\varphi \overset{\Delta}{=} \neg F \neg \varphi. \]

3. Release: Dual operator of until:
   \[ \psi R\varphi \overset{\Delta}{=} \neg (\neg \psi U \neg \varphi). \]

Based on the definition of a run, the semantics of LTL formulas is defined in Figure 2.10. In this definition \( \alpha, i \models \varphi \) means that \( \varphi \) holds in multiset \( M_i \) of run \( \alpha \).

A temporal logic formula \( \varphi \) holds for a run \( \alpha \) if and only if \( \alpha, 0 \models \varphi \). An LTL formula \( \varphi \) holds for a schedule \( s \) with an initial multiset \( M \) if and only if it holds for all valid runs of \( s \) and \( M \). A temporal logic formula holds for a GAMMA program if and only if it holds for its corresponding most general schedule.
\[ \alpha, i \models p \iff |\alpha| > i \text{ and } M_i \models p \quad \text{(for atomic formula } p) \]
\[ \alpha, i \models \neg \varphi \iff |\alpha| > i \text{ and } \alpha, i \not\models \varphi \]
\[ \alpha, i \models \varphi_1 \lor \varphi_2 \iff \alpha, i \models \varphi_1 \text{ or } \alpha, i \models \varphi_2 \]
\[ \alpha, i \models X \varphi \iff |\alpha| > i + 1 \text{ and } \alpha, i + 1 \not\models \varphi \]
\[ \alpha, i \models \varphi_1 \mathbf{U} \varphi_2 \iff \text{there exists a } k \text{ such that } k \geq i, |\alpha| > k, \text{ and } \alpha, k \models \varphi_2 \]
\[ \text{and for all } j \text{ such that } i \leq j < k: \alpha, j \models \varphi_1 \]

---

**Figure 2.10: Semantics of Linear-time Temporal Logic Formulas**

**Example 2.9 (Dining philosophers, Correctness properties)** Any model/solution for this problem should have two basic properties:

1. No two neighboring philosophers eat at the same time. In other words, philosophers cannot share a fork while eating (providing mutual exclusion). This property is expressed as the following set of LTL formulas:

\[
\begin{align*}
G((PE, i) \in \text{room} & \Rightarrow (PE, (i + 1) \mod n) \not\in \text{room} ) \\
& \land \\
& (PE, (i - 1) \mod n) \not\in \text{room} )
\end{align*}
\]

for all \( i: 0 \leq i < n \).

2. Once a philosopher gets hungry (and transfers to the sitting state), s/he eventually finds the opportunity to eat (preventing individual starvation):

\[
G((PS, i) \in \text{room} \Rightarrow F(PE, i) \in \text{room})
\]

for all \( i: 0 \leq i < n \).

Our GAMMA program in Example 2.5, satisfies the first property by construction. The property holds on all runs, because:

1. The property holds in the initial multiset;

2. For all GAMMA rules in the program: First, an individual execution of a rule in a correct state (where no two neighboring philosophers eat) ends in a correct state. Second, two substitutions that allow introduction of neighboring philosophers are not independent; hence, they cannot be executed in parallel.

The two proposed coordinated programs in Example 2.7, also satisfy the first property following the same reasoning. This result could be generalized by defining a refinement relation between schedules that preserves certain temporal properties such as invariants. The mutual-exclusion property is an example of an invariant. [8] contains several possible refinement relations between schedules that could be exploited for this reason.
The second property does not hold for the general GAMMA program. A counter-example is a run presenting three philosophers, with two thinking, sitting, and eating iteratively and the third one sitting and waiting without being served. But, the schedules proposed in Example 2.7 solve the problem of starvation by providing a guarantee for feeding every hungry philosopher (for GAMMA programs with more than one philosopher) after serving finitely many (at most \( n - 1 \)) others.

An interesting observation is that the amount of time after which a philosopher is served seems to be different for the two schedules since the first one serves them in a sequential manner and the second one concurrently; however, we are not able to reason about it at this stage because the model abstracts from quantitative timing.

**Example 2.10 (The elevator system, Correctness properties)** A property of the elevator system is that any request from inside or outside is eventually serviced:

\[
G((\text{extStop} \cdot i), \text{on}) \in M \Rightarrow F((\text{extStop} \cdot i), \text{off}) \in M
\] 
\[
\wedge
\]
\[
((\text{inStop} \cdot i), \text{on}) \in M \Rightarrow F((\text{inStop} \cdot i), \text{off}) \in M
\]

for all \( i: 0 \leq i < \text{MaxFloor} \).

The general GAMMA program for this system (given in Example 2.6) comes short of satisfying this property since, as mentioned there, it allows runs with the elevator just going up and down without servicing any request. However, the given schedule in Example 2.8 satisfies this property by iteratively going up and down and servicing every request in the worst case after visiting \( n - 1 \) other floors at most two times.
Chapter 3

Time

3.1 Introduction

Abadi and Lamport argue in [1] that adding a new state variable named now will extend untimed system semantics for real-time settings in practice. However, the rest of their paper shows, ironically, that this new variable is too different from a normal state variable and fixing the right semantics for it in a system may need relatively long logical formulas (for example 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 are governed by their performance and timeliness specifications, which is a separate quality aspect with respect to the untimed functionality. Thus, it makes sense to specify timing information (requirements) as a separate aspect of the untimed model and observe how it reflects on the semantics of functional and behavioral aspects when composed with their respective models.

This chapter provides a background on our design method and the ways of modelling time to prepare the basis for our move to a timed setting. First, we review our general design philosophy and see how it relates to the timed setting. Subsequently, in Section 3.2, we present the decision points for an extension with time. Section 3.3 concludes this chapter by presenting the choices made on these decision points, as well as their rationale.

Our main motivation in developing a new timed formalism is, first, to support a separation of concerns in the design of real-time systems and, then, exploit this separation in verifying the correctness properties, and finally transform the correct design to the implementation domain. As we mentioned before, the proposed separation of concerns paradigm is meant to support a more focused design method that also supports localized change and even assessment of some models in absence of the others. To support these features in addition to the syntactic separation of models, we have to provide a formal semantics for each meaningful composition of them.
In particular, in the timed-GAMMA/coordination settings the general goals will be realized by the following requirements on our models:

1. Timed-GAMMA should provide a basic model of timed-functionality that observes the performance measures of components. Furthermore, it should show the chaotic behavior of GAMMA programs by allowing all possible timed-behaviors of starting tasks simultaneously or sequentially in any possible order giving them time to execute, preempting them, etc. In other words, the only constraint on the timed-GAMMA program should be observing the execution time of components and beside that, it should be as general as possible. This general model will serve as an ideal starting point for further restrictions using refinement methods or composition with other aspect constraints.

2. Timed-Coordination should only add ordering and synchronization information of the execution of components to the basic functionality model. In other words, the purpose of the timed-coordination language is to execute components in such an order that certain timed correctness requirements are met. Hence, the timed-coordination language does not need to address the time in its syntax (i.e., the syntax does not change) and it only needs to use the timing information (provided by the timed-functionality model) in its semantics. Note that this model of hierarchical semantics for timed-GAMMA/Coordination provides the possibility to substitute the time and/or functionality model with another model and still use the coordination model on top of it.

3. We distinguish between execution-time (performance) modelling of individual components (i.e., GAMMA rules) and end-to-end timing requirements. While the former is a separate aspect model of basic functionalities, the latter is a requirement over the composed behavior. Although timed-GAMMA has to deal with the execution-time measures for its rules, ultimately our design method should provide a way of composing timed-components to meet the end-to-end timing requirements. We will be looking at refinement and similar software construction methods for this purpose in the future.

### 3.2 Modelling Time: Decision Points

The following decision points are to be solved based on the previous requirements:

1. **Periodic vs. sporadic behaviors**: A periodic process is a process that is executed iteratively following a fixed period of time. On the other hand, for a sporadic process, this period is not fixed and only an estimation can be given for the minimum amount of time between two executions of the process. As stated in [30]:

   

---

35
Periodic processes constitute a bulk of computation, while rare sporadic processes seem to be useful only for environment change purposes.

2. Ways of representing time: Time can be represented in a semantics by a point (a single value), an interval, or a set (of points, or intervals). Using points in representing time is the most restrictive method. In contrast, using a set (of either points or intervals) is the most general one in which different ways of a computation (e.g. with disjoint intervals) can be modelled in a natural way.

As the specification of timing may depend on properties of a system (e.g. distribution and replication), it should be possible to represent such dependencies in the specification of timing. To this extent, the above representations of time could be generalized also to functions (with parameters from other domains) associating to a computation either a single value, an interval, or a set of values (set of intervals).

3. Relative vs. absolute time: In presenting semantics for real-time systems, time can be shown relative to some initial time point or relative to the time of the previous state / transition. The first option is referred to as absolute time, whereas the second notion is called relative time. As [19] puts it, in an absolute time setting time contains events while in a relative time setting events constitute time.

4. Dense vs. discrete time: In dense time models, observation of system behavior is done in snapshots with a time taken from a dense set (a set is dense with respect to an ordering \(<\) if between two arbitrary elements \(a, b\) always a third element \(c\) can be found such that \(a < c < b\); for example, the set of rational numbers is dense with respect to the standard \(<\) ordering). On the other hand, discrete time semantics are concerned with digital discrete time-value sets (natural numbers, as a distinguished example).

5. Instantaneous vs. durational actions: An instantaneous action may wait some time before execution but the state change is done without changing time. In this setting, transitions in a real-time system could be partitioned into idle(void), time-changing, and state-changing transitions. On the other hand, in durational semantics, a transition takes time between its beginning and completion and, hence, state-changing transitions may be time-changing as well.

6. Global vs. distributed time: A real-time semantics, has to associate a time to states and/or transitions. This time can be uniquely increased in different parts of the system based on some global clock, even if the system is physically distributed. However, in a distributed time setting, a system may exploit different time systems (distributed clocks) in different parts.
7. **Time stamping states and/or transitions**: To add time to an untimed semantics, a choice must be made between adding time to states, transitions, or to both.

8. **Progress**: Different progress assumptions may be desired due to different uses of the model. For example when compositionality is considered as a goal, allowing relaxed idle transitions (stuttering steps) may represent the time that the processor spends on other parts of computation not specified by the current program. Thus, in this setting, allowing idle steps and relaxing the progress assumption is quite helpful. On the other hand, when verification of a closed model is considered, assuming a kind of progress may be inevitable to prove useful properties.

### 3.3 Decisions and Their Rationale

Considering the previously mentioned requirements and decision points, the following decisions are made to fix the semantics of our model:

1. **Periodic vs. sporadic behaviors**: We have a preference in favor of sporadic process modelling. Assuming periodic execution of atomic actions is considered to be an assumption about their coordination (scheduling), which is to be avoided according to our requirements. After providing a suitable schedule, the whole schedule (or its building blocks) could be dispatched to processors as periodic processes.

2. **Ways of representing time**: A first choice is an interval time representation because we add time to GAMMA for the purpose of estimating execution time (and indirectly other timing properties like latency); it would be hard to have a specific point estimation for execution time. However, a natural extension is to associate (a finite set of) different intervals to an operation to gain the most general modelling schema.

   With respect to the previously mentioned inter-dependencies of the timing domain and other specification domains, extending the interval (set of interval) notation to a function taking a specification of other aspects into account can be another extension to our model. Since we did not define any other aspects by now, this extension is not possible at this stage.

   Another useful extension is to declare the timing information as functions of system state. Because the information for determining the execution time is available when scheduling, this is also a straightforward extension to our model.

3. **Relative vs. absolute time**: Using absolute time will lead to an infinite model for any reactive system presented in timed-GAMMA. Hence, using relative time has a major advantage over absolute time (especially, when considering formal techniques on reasoning). As we see in our model, by
assuming a relative time model, one can derive absolute time models by fixing the origin of time.

4. **Dense vs. discrete time:** Apart from the complexity of reasoning in the verification and refinement area, using dense or discrete time will have no effect on our semantics. Hence, we keep the discussion abstract from the discreteness or denseness of the time domain as much as possible. In order to present benefits and drawbacks of a particular decision on the time domain, we present reasons supporting each side briefly.

Since quantized time is the only time paradigm in computer systems (due to the presence of digital clocks), algorithms assuming that processes have arbitrary integer computation times should be satisfactory for most applications [30]. Also since a discrete time semantics can be closed under scaling the time unit and shifting the origin of time, it can provide sufficient precision. Dealing with discrete time domains sometimes is the solution to make the verification process decidable.

On the other hand, it should be mentioned that there exist some continuous models of real-time systems (even without so-called Zeno behavior) that can not be simulated by any discrete time model [2]. In such systems, scaling the discrete time unit to arbitrary small chunks does not solve the problem, either. Although one may argue that such systems do not exist in reality, they are at least important from a theoretical point of view. Furthermore, in the dense time domain, digitizability theorems show that there always exists a discrete (digital) model of a dense time one that preserves the digital versions of properties that were true for the dense time domain. Hence, a dense time domain is the more general choice than a discrete one.

5. **Global vs. distributed time:** Almost all models studied up to now use a centralized clock for measuring and increasing time globally over the system. Although this assumption is not close to the real world (where providing a single global clock is a problem), it can be rationalized by using clock synchronization algorithms in a distributed platform (in the background, transparent from our modelling). So, we choose a centralized clock model and leave the choice for distributed timing schemes for further extensions to our modelling language.

6. **Instantaneous vs. durational actions:** Instantaneous actions make the modelling semantics more compact and easier to reason about. In durational semantics, the system state during execution would not be predictable and a model for assuring atomicity should be defined. Hence, we use the instantaneous modelling of actions for now.

7. **Time stamping states and/or transitions:** With instantaneous actions and global time, each computation trace shows deterministically the system state (values) with their time (when fixing the time origin, an absolute time), and so, adding a time stamp to states would not be useful.
in this case. Hence, we just use time stamping on transitions. For distributed time modelling, however, time stamping states may make sense for observing causal relations and for synchronization purposes.

8. **Progress**: Due to our design philosophy, we do not want to impose any restriction on our functionality and coordination model unless explicitly specified. Hence, we assume a general progress assumption in which idling is generally allowed. To restrict idling in the behavior of schedules explicitly, we introduce urgency operators.
Chapter 4

Timed-GAMMA, Coordination, and Reasoning

4.1 Introduction

The general design philosophy and the decisions made in the previous chapter lead to the syntax and semantics of timed-GAMMA and our timed-Coordination language. To preserve the orthogonality between models, we want to extend the GAMMA and coordination model in such a way that an un-timed GAMMA program or an un-timed coordination expression can be interpreted in the timed setting in a meaningful way. In fact, the syntax of our coordination language does not change and only its semantics is updated. This will provide both a kind of backward compatibility and the freedom for the designer to be free from timing concerns when designing the functionality and behavior aspects.

In this chapter, first we present the syntax and semantics of timed-GAMMA in Section 4.2. Then, the semantics of timed-Coordination is defined in Section 4.3 based on the new timed functionality model. Finally in Section 4.5, the method of reasoning is extended to support quantitative reasoning about timed functionality and timed behavior. In order to support such reasoning in Section 4.4 an urgency operator is introduced.

4.2 Timed-GAMMA

4.2.1 Timed-GAMMA Syntax

In this subsection, the syntax of GAMMA programs is extended with timeliness properties. Following [4, 24], the change in syntax is done in such a way that a valid GAMMA program is also a valid Timed-GAMMA program.
TimedProgram ::= Program Timing
Timing ::= ε | TConstraint Timing
TConstraint ::= TRuleName = Interval
Interval ::= [Time, Time] | (Time, Time]
         | [Time, Time∞] | (Time, Time∞)

Figure 4.1: Timed-GAMMA Syntax

The syntax of Timed-GAMMA is presented in Figure 4.1. The non-terminal
Program represents an un-timed GAMMA program as presented in Figure 2.3
in Section 2.3.2. Timing constraints are added to the GAMMA program definition
to represent an estimation of the rule execution time. A timing, defined by the
non-terminal TConstraint, relates an interval (a convex set of time points) to a
rule via its rule name. Such an interval represents the minimum and maximum
time needed to perform a rule computation. This time is relative to the point
from which the rule is scheduled for execution (i.e., when the previous transition
is finished). To emphasize that an un-timed program is indeed a timed program,
the timing part is isolated via non-terminal Timing with the possibility of being
the empty string ε.

Definition 4.1 (Basic time domain) The basic time domain is a set denoted
by Time that contains an element denoted 0 and on which a total ordering <
with least element 0 and an addition function + with unit element 0 are defined.
We do not need to have any more assumptions on Time, until we apply the
construction of schedules or automatic verification in the future.

To model unspecified upper bounds of intervals we introduce a new element
to the basic time domain, denoted by ∞, and refer to the time domain as:
Time∞ ≡ Time∪{∞}. We extend the ordering relation and addition function
on Time∞ such that for all t ∈ Time, t < ∞, t + ∞ = ∞ + t = ∞, and
∞ + ∞ = ∞.

Bounds of intervals are elements taken from these time domains as described
by the non-terminal Interval. If there is no timing estimation specified for a
rule (as is the case for un-timed specifications), it is assumed to be [0, ∞), that
is, an arbitrary execution time. We refer to the lower and upper bound of the
interval I with lb(I), and ub(I), respectively. We use addition and multiplication
of constants to intervals (t + I, and t * I, where t is an arbitrary element of the
time domain) to represent addition and multiplication of its bounds.

Definition 4.2 (In-bound operator) We define that t ∈ Time is in-bound
with respect to the interval I, denoted by t ≪ I if and only if t ≤ lb(I) ∀ t ∈ I.
\[
(RuleTerm) \quad \exists \psi \in r, M, \psi \preceq r \\
\quad \langle r, M, \emptyset \rangle \rightarrow 1
\]

\[
(Rulesched) \\
\quad M, \psi \preceq r \\
\quad \langle r, M, \emptyset \rangle \rightarrow 1 \langle r, M, [\sigma \circ \delta] : r.I \rangle
\]

where for a rule \( r = \text{LE}xp \rightarrow \text{RE}xp \leftrightarrow \text{Cond} \), \( \sigma = \psi[\text{RE}xp]/\psi[\text{LE}xp] \)

\[
(RuleComp) \\
\quad t \in I \\
\quad \langle r, M, [\sigma @ t : I] \rangle \rightarrow 0 \langle r, M[\sigma], \emptyset \rangle
\]

\[
(TimePass) \\
\quad t + t' \prec I \\
\quad \langle r, M, [\sigma @ t : I] \rangle \rightarrow 0 \langle r, M, [\sigma @ t + t' : I] \rangle \quad t' > 0
\]

---

**Figure 4.2**: Timed-GAMMA Semantics: Basic Functionality

4.2.2 Timed-GAMMA Semantics

Following the style of the semantics of GAMMA, the semantics of Timed-GAMMA is presented in two parts. The first part, presented in Figure 4.2, shows the basic timed-computation and termination of a GAMMA rule. In the given semantics, a timed state \( \langle r, M, T \rangle \) consists of a rule \( r \), a multiset \( M \), and a multiset of scheduled computation tasks \( T \). A task \( \sigma @ t : I \) is the substitution \( \sigma \) 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 \( I \) derived from the rule form which \( \sigma \) originates. As shown in the syntax of timed-GAMMA, after defining the rules of a GAMMA program, an estimation of the execution times of rules is presented in the *Timing* definitions. To refer to an interval \( I \) associated with a rule named \( r \), we use \( r.I \). As mentioned before, if there is no interval defined for a rule with the name \( r \), \( r.I \) results in \([0, \infty)\). For the time being, we assume that \( r.I \) works as a function returning the interval representing the estimation of the execution time of a rule, if available, or otherwise \([0, \infty)\). As mentioned before, 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.

The transition labels in the given semantics are either of the form \( \rightarrow \) (or \( \rightarrow_1 \)), where \( t \) can range over the basic time domain \( \text{Time} \) or \( \rightarrow \) (or \( \rightarrow_1 \)), just like in the un-timed semantics where \( \sigma \) ranges over multiset substitutions.

A basic timed-computation is divided into three phases:
1. First, scheduling a computation task based on a single rule. In the semantics, the scheduling of a task is indicated by a transition $\xrightarrow{\tau} 1$. The rule (RuleSched) shows how a task is scheduled. If there is an enabling valuation for a rule, then the corresponding task is added to the scheduled tasks with zero elapsed processing time. Note that in the single-rule execution semantics, the task multiset should be empty in order to schedule a new task. In other words, since at this level of semantics we want to deal with basic functionalities, we cannot introduce several parallel tasks at one time.

2. Second, passage of time (by spending processor time on a computation). Passage of a certain amount of time $t > 0$ is in the semantics indicated by a transition $\xrightarrow{-1}$. Rule (TimePass) expresses that for a scheduled task a certain amount of processing time can be spent up to the limits of the scheduled task as specified by the interval $(t + t' < I)$. We cannot use the simpler conditions $t < ub(I)$ or $t \leq ub(I)$ since they would be problematic with right-closed and right-open intervals, respectively. We do not allow passage of zero time, because it is of no interest in our setting and to distinguish time pass rules from the task scheduling rule.

3. Finally, performing (committing) a computation. The commitment of a certain task is indicated by a transition $\xrightarrow{\sigma}$ where $\sigma$ is the substitution associated with that task. Rule (RuleComp) specifies that when a task has received enough processing time $(t \in I)$, then it can be applied to the multiset and removed from the current tasks.

This division 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) shows that a rule will terminate if there is no enabling valuation for it and if the current task multiset is empty (there is no scheduled task being executed).

The following basic properties of the basic timed-computation can be proven. Commitment of a task extends the task multiset with precisely one task. Time passage only changes the timing part of the scheduled task.

Lemma 4.1 For any rule $r$, multisets $M$ and $M'$, task multisets $T$ and $T'$, substitution $\sigma$, and $t \in \text{Time}$ such that $t > 0$:

1. if $(r, M, T) \xrightarrow{0} (r, M', T')$, then $M' = M$, $T = \emptyset$, and $T' = [\sigma' @ 0 : r, I]$ for some substitution $\sigma'$;
2. if $(r, M, T) \xrightarrow{-1} (r, M', T')$, then $M' = M[\sigma]$ and $T' = \emptyset$;
3. if $(r, M, T) \xrightarrow{-1} (r, M', T')$, then $M' = M$. 

43
Proof. The proofs of these propositions follow easily from the semantics in Table 4.2.

The second part of the semantics, given in Figure 4.3, specifies the general chaotic behavior of timed-GAMMA programs by composing the behavior of rules in all possible orders and also allowing idling transitions. A state \(\langle R, M, T \rangle\) consists of a program \(R\), a multiset \(M\), and a multiset of scheduled tasks \(T\). To show how independent tasks can be scheduled simultaneously in the Timed-GAMMA chaotic execution model of Figure 4.3, we need to define the independence of task multisets.

**Definition 4.3** Composed substitution of a task multiset denoted by \(\text{Subst}(T)\) is a multiple substitution resulting from composing all substitutions in \(T\) and is defined as:

1. For an empty task multiset: \(\text{Subst}(\langle \rangle) = \langle \rangle / \langle \rangle\)
2. For an arbitrary singleton task multiset: \(\text{Subst}(\langle \sigma \rangle : I) = \sigma\)
3. For two arbitrary task multisets: \(\text{Subst}(T_1 \boxplus T_2) = \text{Subst}(T_1), \text{Subst}(T_2)\)

**Definition 4.4 (Independent task multisets)** For an arbitrary multiset \(M\), we define that two arbitrary task multisets \(T_1\) and \(T_2\) are independent with respect to \(M\) (denoted by \(M \models T_1 \Leftrightarrow T_2\)) if and only if, their respective composed substitutions are independent, that is \(M \models \text{Subst}(T_1) \Leftrightarrow \text{Subst}(T_2)\). We call a task multiset \(T\) consistent with respect to a multiset \(M\), if and only if \(M \models T \Leftrightarrow \langle \rangle\).

**Corollary 4.2** For all task multisets \(T_1\) and \(T_2\) and an arbitrary multiset \(M\), \(T_1\) and \(T_2\) are independent with respect to \(M\) if and only if \(T_1 \boxplus T_2\) is consistent with respect to \(M\).

**Lemma 4.3** For arbitrary task multisets \(T\) and \(T'\) and multiset \(M\); if \(T \boxplus T'\) is consistent w.r.t. \(M\), then \(T'\) is consistent w.r.t. \(M[\text{Subst}(T)]\).

*Proof.* Suppose that \(T = [\sigma_0 \circ \ell_0 : I_0, \ldots, \sigma_n \circ \ell_n : I_n]\) and \(T' = [\sigma_{n+1} \circ \ell_{n+1} : I_{n+1}, \ldots, \sigma_m \circ \ell_m : I_m]\). Then, according to Definitions 2.7 and 4.4 we have

\[
\bigcup_{i=0}^{n} \text{read}(\sigma_i) \circ \bigcup_{i=0}^{n} \text{take}(\sigma_i) \subseteq M. 
\]

It follows from definition of \(\circ\) that \(\bigcup_{i=0}^{n} \text{take}(\sigma_i) \subseteq (M \bigcirc \bigcup_{i=n+1}^{m} \text{take}(\sigma_i)) \circ \bigcup_{i=n+1}^{m} \text{put}(\sigma_i)\). Hence (and due to consistency of \(T \boxplus T'\)), we have \(\bigcup_{i=0}^{n} \text{take}(\sigma_i) \subseteq M[\text{Subst}(T')]\). 

In Figure 4.3, the semantics of timed-GAMMA programs is presented. In this semantics, rule (ProgTerm) extends the termination of a single rule to
(ProgTerm) $\forall r \in R \quad \langle r, M, T \rangle \vdash_1 \langle r, M', T \rangle$

(ProgComp0) $r \in R \quad \langle r, M, T \rangle \vdash_1 \langle r, M', T \rangle$

(ProgComp1) $\langle r, M, T \rangle \vdash_1 \langle r, M', T \rangle$

(ProgTime0) $r \in R \quad \langle r, M, T \rangle \vdash_1 \langle r, M', T \rangle$

(ProgTime1) $\langle r, M, T \rangle \vdash_1 \langle r, M', T \rangle$

(ProgSched) $\langle r, M, T \rangle \vdash_1 \langle r, M, T \rangle$

\[ M \models [\sigma @ I] \models T \]

Figure 4.3: Chaotic Behavior of Timed-GAMMA Program
termination of a program. It follows from this rule and rule (RuleTerm) of Figure 4.2 that Timed-GAMMA programs can only terminate if the task multiset is empty.

Lemma 4.4 For all timed-GAMMA programs R and multisets M, if \((R, M, T)\) then \(T = [\emptyset]\).

Proof. A program \(R\) can terminate if and only if for all its rules \(r\), \((r, M, T)\) terminate. Termination of a rule \(r\) can only be due to (RuleTerm) which means that \(T = [\emptyset]\).

The rules (ProgComp0) and (ProgComp1) specify how a timed-GAMMA program can perform single or parallel computations. Along with these rules (ProgTime0) and (ProgTime1) specify the spending of time on the execution of single and parallel computations. The above four rules provide the same kind of abstraction from the true concurrency level that we also have in untimed GAMMA and extend it to the timed setting. In particular, by allowing a subset of tasks to spend time on their computations, we model the cases were there are not enough resources for true concurrency of all tasks. As before, we do not allow passage of zero time. 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. The rule (ProgIdle) states that a program can make an arbitrary idle transition without changing its state. This rule helps to keep the timed-GAMMA semantics on the high level of abstraction and allows stuttering steps that can be helpful for compositionality. Lemma 2.10 which holds in the untimed setting carries over to the timed setting. It states that if a program can perform a computation, then a larger program can do the same, as well.

Lemma 4.5 If a program \(R\) can perform an arbitrary transition, i.e., \((R, M, T) \rightarrow (R, M', T')\), then for an arbitrary rule \(r\), \((R \cup \{r\}, M, T) \rightarrow (R \cup \{r\}, M', T')\).

Proof. By induction on the depth of the proof for \((R, M, T) \rightarrow (R, M', T')\). Similar to the untimed proof, we have to analyze the \(\chi\) transition. In each case we will either end up in a single rule transition (from which by application of the same semantics rule we are done) or a rule and a program. In the latter case, we have to apply the induction step on the program transition and then use the same semantics rule to combine the new extended program and the rule transition. For example if \(\chi\) is a substitution, then the transition is due to one of the following rules:

1. (ProgComp0): Then, \(T = [\sigma \oplus t : I] \boxplus T'\) and there exists a rule \(r \in R\), such that \((r, M, [\sigma \oplus t : I]) \rightarrow (r, M', [\emptyset])\). Then, according to the same rule (ProgComp0), \((R \cup \{r\}, M, [\sigma \oplus t : I] \boxplus T') \rightarrow (R \cup \{r\}, M', T')\).
2. \textbf{(ProgComp1):} Then, there exist substitutions $\sigma_1$ and $\sigma_2$, multisets $M_1$ and $M_2$, and task sets $T_1$, $T_1'$, $T_2$, and $T_2'$, such that $T = T_1 \uplus T_2$, $T' = T_1' \uplus T_2'$, $\sigma = \sigma_1 \cup \sigma_2$, $M' = M[\sigma_1 \cup \sigma_2]$, $(R, M, T) \rightarrow_2 \langle R, M_1, T_1' \rangle$ and $(R, M, T_2) \rightarrow_2 \langle R, M_2, T_2' \rangle$. Then, by induction, $(R \cup \{r\}, M, T_1) \rightarrow_1 (R \cup \{r\}, M_1, T_1')$ and $(R \cup \{r\}, M, T_2) \rightarrow_1 (R \cup \{r\}, M_2, T_2')$. According to rule \textbf{(ProgComp1)}, $(R \cup \{r\}, M, T_1) \rightarrow_1 (R \cup \{r\}, M_{1'}, T_1')$, i.e., $(R \cup \{r\}, M, T) \rightarrow_1 (R \cup \{r\}, M', T')$.

Some properties of the given semantics can be proved to show that the formal semantics follows the informal intuition behind it. These properties reflect the same intuitions as were given for the basic timed-computation semantics in Lemma 4.6.

\textbf{Lemma 4.6} For any set of rules $R$, multisets $M$ and $M'$, task multisets $T$ and $T'$, substitution $\sigma$, and $t \in \text{Time}$ such that $t > 0$:

1. if $(R, M, T) \rightarrow (R, M', T')$, then $M' = M$ and $T' = [\sigma_t^{\uplus : I}] \uplus T$ for some substitution $\sigma_t$ and interval $I$;

2. if $(R, M, T) \rightarrow (R, M', T')$, then $M' = M[\sigma]$ and $\text{Subst}(T'), \sigma = \text{Subst}(T)$;

3. if $(R, M, T) \rightarrow (R, M', T')$, then $M = M'$ and $\text{Subst}(T) = \text{Subst}(T')$.

\textit{Proof.}

1. The 0 transition can only be due to \textbf{(ProgSched)}. Thus, the case trivially follows from this rule.

2. By an induction on the depth of the proof for $\sigma$ transition. Suppose that the transition has a proof of length 1, then it can only be due to \textbf{(ProgComp0)}. Hence, the case holds trivially for the induction base. For the proofs of length $n > 1$, the proof is due to \textbf{(ProgComp1)}. Thus, the case holds by applying induction hypothesis to the premises of this rule.

3. Similar to item 2.

The following proposition is an example of such a property showing that the chaotic behavior of a GAMMA program only allows consistent multisets of tasks to be scheduled simultaneously.

\textbf{Proposition 4.7} For all timed states $S = (s_1, M_1, T_1)$ and $T = (s_2, M_2, T_2)$, if there exists a transition $\gamma$ such that $S \rightarrow T$ and $T_1$ is consistent with respect to $M_1$, then $T_2$ is consistent with respect to $M_2$. 

47
Proof. By induction on the depth of the proof of \( S \rightarrow^* T \). We only prove that each rule maintains consistency.

1. (ProgComp0) Suppose that \( S = (R, M, [\sigma @ t : I] \downarrow T) \) and \( S' = (R, M', T) \). Suppose furthermore that \([\sigma @ t : I] \downarrow T\) is consistent w.r.t \( M \). We have to prove that \( T \) is consistent w.r.t. \( M' \). From this, by Lemma 4.1.2, it follows that \( M' = M[\sigma] \). By Lemma 4.3, consistency of \( T \) w.r.t. \( M' \) follows from consistency of \([\sigma @ t : I] \downarrow T\) w.r.t. \( M \) and \( M' = M[\sigma] = M[Subst([\sigma @ t : I])] \).

2. (ProgComp1) Suppose that \( T_1 \uplus T_2 \) is consistent w.r.t. \( M \). From this it follows that \( T_1 \) is consistent w.r.t. \( M \) and that \( T_2 \) is consistent w.r.t. \( M \). Then, by induction, we obtain the consistency of \( T_1 \) w.r.t. \( M_1 \) and \( T_2 \) w.r.t. \( M_2 \). It can be proven that \( M_1 = M[\sigma_1] \) and \( M_2 = M[\sigma_2] \). Also, by Lemma 4.6.2, \( Subst(T_1) = Subst(T_1'), \sigma_1 \) and \( Subst(T_2) = Subst(T_2'), \sigma_2 \). According to Lemma 2.6, \( M[\sigma_1] \models Subst(T_1') \Rightarrow Subst(T_2'), \sigma_2 \) and by application of the same lemma once more, \( (M[\sigma_1])[\sigma_2] \models Subst(T_1') \Rightarrow Subst(T_2') \). But using Lemma 2.5 we have, \( M[\sigma_1, \sigma_2] \models Subst(T_1') \Rightarrow Subst(T_2') \). This means that \( M' \models T_1' \Rightarrow T_2' \).

3. (ProgTime0) Suppose that \([\sigma @ t : I] \downarrow T\) is consistent w.r.t. \( M \). We have to prove that \([\sigma @ t + t' : I] \downarrow T\) is consistent w.r.t. \( M \). This follows immediately from the definitions by the observation that \( Subst([\sigma @ t + t' : I]) = Subst([\sigma @ t : I]) \).

4. (ProgTime1) The consistency of \( T_1' \uplus T_2' \) w.r.t. \( M \) follows immediately from the consistency of \( T_1 \uplus T_2 \) w.r.t. \( M \) using Lemma 4.6.3.

5. (ProgSched) Consistency of \([\sigma @ 0 : I] \downarrow T\) w.r.t. \( M \) follows from \( M \models [\sigma @ 0 : I] \Rightarrow T \).

6. (ProgIdle) Obviously, consistency of \( T \) w.r.t. \( M \) follows from consistency of \( T \) w.r.t. \( M \).

As mentioned in the previous chapter, the intention of the semantics given here is different from the one in release time, deadline semantics (e.g. [20, 18]), since timing constraints in that case specify the minimum and the maximum amount of time an enabled transition should wait before being taken. However, in our case, the presented estimations are only important when the rule is scheduled for execution. In other words, the timing estimation in our model is based on individual component executions and usually due to chaotic behavior of GAMMA program or later due to a coordination strategy, an enabled rule may wait (without being scheduled) more than its maximum estimated execution time. We believe that specification of release time and deadline for a rule, is
also an important issue but it should be specified separately and proved correct through a coordination strategy. This is shown in the later examples.

**Example 4.1 (Dying dining philosophers)** Recall the previous definition of the classic dining philosophers problem (Example 2.5); the problem definition is extended with timing properties as follows.

Each philosopher, after spending an arbitrary amount of time (between 0 and $t_{\text{think}}$) on philosophical thoughts, gets hungry, sits on his/her chair and tries to get hold of both forks on his/her right and left. Unfortunately, one (or both) of these forks may be already in use by a colleague (colleagues) and s/he should wait for the colleague(s) to finish eating. Eating also takes an arbitrary time, between 0 and $t_{\text{eat}}$. After the philosopher finishes eating, s/he is ready to leave the table. It is assumed that when a fork is released it can be instantaneously available to the one waiting for it (washing forks takes no time).

A general statement of the problem can be found in [25]. The timed-GAMMA specification of the dying dining philosophers is:

\[
\text{DyingDiningPhilosophers} = \\
\begin{align*}
\text{Sit} &= (P, i) \rightarrow (PS, i), \\
\text{Eat} &= (PS, i), (F, i), (F, (i + 1) \mod n) \rightarrow (PE, i), \\
\text{Leave} &= (PE, i) \rightarrow (P, i), (F, i), (F, (i + 1) \mod n)
\end{align*}
\]

To get a better intuition of the GAMMA semantics, we take a concrete example of the dying dining philosophers with $n = 4, t_{\text{think}} = 2, t_{\text{eat}} = 3$. Figure 4.4 depicts examples of a timed-GAMMA program run (sequence of computations). In figure 4.4 (a), the substitution names stand for the following substitutions: $\sigma_{\text{think}1} = [(PS, 0)\ ||(F, 0)]$, $\sigma_{\text{think}2} = [(PS, 2)\ ||(F, 2)]$, and the initial multiset is $M = [(P, 0), \ldots, (P, 3), (F, 0), \ldots, (F, 3)]$. The depicted run could be executed according to the GAMMA semantics as follows (we show an acronym of the rule names allowing each transition):

\[
\begin{align*}
\langle R, M, \Box \rangle \\
\xrightarrow{0} &\quad \text{(RuleSched), (ProgSched)}, M \models [\sigma_{\text{think}0} \circ 0 : T_{\text{Sit}}] \triangleright \Box \\
\xrightarrow{1} &\quad \text{(TimePass), (ProgTime0)} \\
\langle R, M, [\sigma_{\text{think}0} \circ 0 : T_{\text{Sit}}] \rangle \\
\xrightarrow{0} &\quad \text{(RuleSched), (ProgSched)}, M \models [\sigma_{\text{think}2} \circ 0 : T_{\text{Sit}}] \triangleright [\sigma_{\text{think}0} \circ 1 : T_{\text{Sit}}] \\
\xrightarrow{1} &\quad \text{(TimePass), (ProgTime0), (ProgTime1)} \\
\langle R, M, [\sigma_{\text{think}0} \circ 1 : T_{\text{Sit}}, \sigma_{\text{think}2} \circ 0 : T_{\text{Sit}}] \rangle \\
\xrightarrow{0} &\quad \text{(RuleSched), (ProgSched)}, M \models [\sigma_{\text{think}0} \circ 2 : T_{\text{Sit}}, \sigma_{\text{think}2} \circ 1 : T_{\text{Sit}}] \\
\xrightarrow{1} &\quad \text{(RuleComp), (ProgComp0), (ProgComp1)} \\
\langle R, M, [\sigma_{\text{think}0}, \sigma_{\text{think}2}], \Box \rangle
\end{align*}
\]
Now, by starting from $M' = M[\sigma_{\text{think}1}, \sigma_{\text{think}2}] = [(PS, 0), (P, 1), (PS, 2), (F, 3), (F, 0), \ldots, (F, 3)]$, we can execute the run in Figure 4.4 (b) using the following transitions. We do not mention the rule name, for transitions similar to previous runs. Substitutions present in the figure stand for $\sigma_{\text{eat}1} = [(PE, 0)]/[(PS, 0), (F, 0), (F, 1)], \sigma_{\text{eat}2} = [(PE, 2)]/[(PS, 2), (F, 2), (F, 3)]$:

\[
\begin{align*}
&\{R, M', \emptyset\} \\
&\xrightarrow{0} \{R, M', [\sigma_{\text{eat}1} @ 0 : T_{\text{eat}}]\} \\
&\xrightarrow{1} \{R, M', [\sigma_{\text{eat}2} @ 1 : T_{\text{eat}}]\} \\
&\xrightarrow{0} \{R, M', [\sigma_{\text{eat}2} @ 1 : T_{\text{eat}}, \sigma_{\text{eat}1} @ 0 : T_{\text{eat}}]\} \\
&\xrightarrow{1} \\
\end{align*}
\]
In the third example, we start from the same multiset and substitutions as before \( M' = M'_{\text{thinks}} \), \( \sigma_{\text{eal}} = [(PE,0)]/[(PS,0), (F,0), (F,1)] \), \( \sigma_{\text{eal2}} = [(PE,2)]/[(PS,2), (F,2), (F,3)] \) and show how the run in Figure 4.4 (c) can be executed using the following transitions:

\[
\begin{align*}
\{R, M', [,]\} \\
\xrightarrow{0} \quad \text{RuleSched}, (\text{ProgSched}), M' = [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}] \times [\emptyset] \\
\{R, M', [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}]\} \\
\xrightarrow{2} \quad \text{RuleSched}, (\text{ProgSched}), M' = [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}, \sigma_{\text{eal2}} \@ 0 : T_{\text{eal}}] \\
\{R, M', [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}, \sigma_{\text{eal2}} \@ 2 : T_{\text{eal}}]\} \\
\xrightarrow{2} \quad \text{RuleComp}, (\text{ProgComp0}) \\
\{R, M', [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}, \sigma_{\text{eal2}} \@ 3 : T_{\text{eal}}]\} \\
\xrightarrow{3} \quad \text{RuleComp}, (\text{ProgComp0}) \\
\{R, M', [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}, \sigma_{\text{eal2}} \@ 3 : T_{\text{eal}}]\} \\
\xrightarrow{3} \quad \text{RuleComp}, (\text{ProgComp0}) \\
\{R, M', [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}, \sigma_{\text{eal2}} \@ 3 : T_{\text{eal}}]\} \\
\xrightarrow{3} \quad \text{RuleComp}, (\text{ProgComp0}) \\
\{R, M', [\sigma_{\text{eal}} \@ 0 : T_{\text{eal}}, \sigma_{\text{eal2}} \@ 3 : T_{\text{eal}}]\} \\
\end{align*}
\]

This execution presents the abstract nature of our parallelism in that it allows sequential execution (or preemption and interleaving) of parallel execution which may be due to shortage of hardware resources.

**Example 4.2 (The elevator system, Timing)** Suppose that the following timing information is given about the elevator system in Example 2.6:

- Requests happen instantaneously (or at least do not take any time from our system).
- Going up and down between floors takes \( \text{StepTime} \) for each floor. Stopping at (starting from) each floor and opening (closing) the door takes \( \text{StartTime} \).
- The elevator will be loaded/unloaded within \( \text{MinService} \) and \( \text{MaxService} \) amount of time, depending on the number of people and goods waiting for it.
(TCoordRuleTerm) 
\( \frac{\langle r, M, \emptyset \rangle \sqrt{1}}{\langle r, M, \emptyset \rangle \sqrt{1}} \)

(\text{TCoordSched}) \quad \frac{\langle r, M, \emptyset \rangle \rightarrow_1 \langle r, M, [\sigma \oplus t : I] \rangle}{\langle r, M, \emptyset \rangle \rightarrow_0 \langle r[\sigma \oplus 0 : I], M, [\sigma \oplus 0 : I] \rangle}

(\text{TCoordComp}) \quad \frac{\langle r, M, [\sigma \oplus t : I] \rangle \rightarrow_1 \langle r, M', \emptyset \rangle}{\langle r[\sigma \oplus t : I], M, [\sigma \oplus t : I] \rangle \rightarrow \langle \text{skip}, M', \emptyset \rangle}

(\text{TCoordTimePass}) \quad \frac{\langle r, M, [\sigma \oplus t : I] \rangle \rightarrow_1 \langle r, M, [\sigma \oplus t + t' : I] \rangle}{\langle r[\sigma \oplus t : I], M, [\sigma \oplus t : I] \rangle \rightarrow \langle r[\sigma \oplus t + t' : I], M, [\sigma \oplus t + t' : I] \rangle}

(\text{TCoordIdle0}) \quad \frac{\langle r, M, \emptyset \rangle \rightarrow \langle r, M, \emptyset \rangle}{\langle r[\sigma \oplus t : I], M, [\sigma \oplus t : I] \rangle \rightarrow \langle r[\sigma \oplus t : I], M, [\sigma \oplus t : I] \rangle}

(\text{TCoordIdle1}) \quad \frac{\langle r, M, \emptyset \rangle \rightarrow \langle r, M, \emptyset \rangle}{\langle r[\sigma \oplus t : I], M, [\sigma \oplus t : I] \rangle \rightarrow \langle r[\sigma \oplus t : I], M, [\sigma \oplus t : I] \rangle}

Figure 4.5: Timed-Coordination Language Semantics (Part1): Basic computation and termination

The following timed-GAMMA program expresses this information:

\[
\text{ElevatorSystem} = \{ \text{inRequest, extRequest, moveUp, moveDown,} \}
\{ \text{close, open, load, unload} \}
\]

\[
\text{T_inRequest} = T_{\text{extRequest}} = [0, 0]
\]

\[
\text{T_moveUp} = T_{\text{moveDown}} = [\text{StepTime, StepTime}]
\]

\[
\text{T_close} = T_{\text{open}} = [\text{StartTime, StartTime}]
\]

\[
\text{T_load} = T_{\text{unload}} = [\text{MinService, MaxService}]
\]

Note that in the above program we dispensed with re-stating the rule part of the GAMMA program and abbreviated them by their respective rule names. Also, for equal intervals we used a shorthand to combine the timing of different rules.

### 4.3 Timed-Coordination

In this part, without changing the syntax of the coordination language given before, we extend its semantics so that it also works for the timed setting. It brings the benefit of orthogonality in our design, in that the design process
for the coordination aspect is completely separated from using real-time constraints, changing them, or even waving them. Due to this feature of our timed-coordination language, we can adopt schedules of previous examples without any change on the syntax and the new semantics takes care of timing associated with atomic rules automatically. Of course, it remains to be verified whether such a schedule satisfies the desired timing properties.

Figure 4.5 shows the first set of semantic rules for the timed-coordination language. These rules link the semantics of single rule execution to the semantics of schedules (coordination terms) and introduce the idling transition (in the same way as the idling of timed-GAMMA programs). In the semantics there is a tight relationship between coordination terms and scheduled tasks. For example, in the execution of schedule \( s || (s ; t) \), we need to distinguish between the two tasks instantiated from the parallel component \( s \) and the sequential component \( s \), since starting \( t \) should be synchronized with commitment of the latter, while the former has nothing to do with execution of the task instantiated by \( t \). Hence, we attach scheduled tasks to its the syntactic expression they originate from (for example, \( r[s \sigma @t ; I] \)).

So, in the given semantics the state \( (s, M, T) \) 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 the timed-GAMMA semantics. 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.

In Figures 4.6 and 4.7, the second part of timed-coordination semantics is presented. In these figures, the transition \( \lambda \) denotes passage of time, scheduling of a task or performing a computation on a substitution \( \sigma \) (\( \chi \) is a variable that ranges over the time domain plus substitutions). According to the first part of the coordination language semantics, the schedule terms present in these figures can be schedules annotated by their corresponding tasks.

Rules (TRC0) to (TRC3) define the semantics for the rule-conditional operator. Expression \( r \land s[t] \) can schedule a task when either the condition rule \( r \) is enabled and the first argument \( s \) can schedule a task (TRC0), or when \( r \) terminates (is disabled at the moment) and \( t \) can schedule a task (TRC1). Obviously, it terminates when none of the above cases are possible. The semantics of the sequential composition operator is defined by rules (TS0), (TS1), and (TS2). Rules (TP0) to (TP3) specify the semantics of the parallel composition operator. In particular, rules (TP0) and (TP1) specify how two sides of a parallel composition can evolve individually in time. Checking independence of task multisets are necessary due to possible scheduling of new tasks by each side. Rule (TP2) is meant to represent true concurrency in which processor time is spent on two different task multisets. Zero time pass is excluded in this rule since, in our semantics, it represents scheduling new task and specifying simultaneous task scheduling is of no particular interest for us, provided that it calls for consistency check of task multisets. Rule (TP3) specifies simultaneous commitment of two or more tasks. The side condition of these rules assures
Figure 4.6: Timed-Coordination Language Semantics (Part 2): Sequential and Rule-conditional Operators
Figure 4.7: Timed-Coordination Language Semantics (Part 2, Contd.): Parallel Composition, and Recursion Operators

\[
\begin{align*}
\textbf{(TP0)} & \quad \frac{}{\langle s_1, M, T_1 \rangle \rightarrow \langle s'_1, M', T'_1 \rangle} \quad \frac{}{\langle s_1 \parallel s_2, M, T_1 \parallel T_2 \rangle \rightarrow \langle s'_1 \parallel s_2, M', T'_1 \parallel T'_2 \rangle} \quad M' \models T'_1 \triangleright T_2 \\
\textbf{(TP1)} & \quad \frac{}{\langle s_2, M, T_2 \rangle \rightarrow \langle s'_2, M', T'_2 \rangle} \quad \frac{}{\langle s_1 \parallel s_2, M, T_1 \parallel T_2 \rangle \rightarrow \langle s'_1 \parallel s_2, M', T'_1 \parallel T'_2 \rangle} \quad M' \models T_1 \triangleright T_2 \\
\textbf{(TP2)} & \quad \frac{}{\langle s_1, M, T_1 \rangle \rightarrow \langle s'_1, M', T'_1 \rangle} \quad \frac{}{\langle s_2, M, T_2 \rangle \rightarrow \langle s'_2, M', T'_2 \rangle} \quad \frac{}{\langle s_1 \parallel s_2, M, T_1 \parallel T_2 \rangle \rightarrow \langle s'_1 \parallel s_2, M', T'_1 \parallel T'_2 \rangle} \quad t > 0 \\
\textbf{(TP3)} & \quad \frac{}{\langle s_1, M, T_1 \rangle \rightarrow \langle s'_1, M', T'_1 \rangle} \quad \frac{}{\langle s_2, M, T_2 \rangle \rightarrow \langle s'_2, M', T'_2 \rangle} \quad \frac{}{\langle s_1 \parallel s_2, M, T_1 \parallel T_2 \rangle \rightarrow \langle s'_1 \parallel s_2, M, T'_1 \parallel T'_2 \rangle} \quad \frac{s_1 \parallel s_2, M, T_1 \parallel T_2}{s'_1 \parallel s_2, M, T'_1 \parallel T'_2} \\
\textbf{(TP4)} & \quad \frac{}{\langle s_1, M, T \rangle \rightarrow \langle s_2, M, T \rangle} \quad \frac{}{\langle s_1 \parallel s_2, M, T \rangle \rightarrow \langle s_1 \parallel s_2, M, T \rangle} \quad \frac{s_1 \parallel s_2, M, T}{s'_1 \parallel s_2, M, T'} \quad \frac{s_1 \parallel s_2, M, T}{s'_1 \parallel s_2, M, T'} \\
\textbf{(TR0)} & \quad \frac{s[\mu z.s/x], M, T \rightarrow \langle s', M', T' \rangle} {\mu z.s, M, T \rightarrow \langle \mu z.s', M', T' \rangle} \\
\textbf{(TR1)} & \quad \frac{s[\mu z.s/x], M, T \rightarrow \langle s', M', T' \rangle} {\mu z.s, M, T \rightarrow \langle \mu z.s', M', T' \rangle} \quad \frac{s[\mu z.s/x], M, T \rightarrow \langle s', M', T' \rangle} {\mu z.s, M, T \rightarrow \langle \mu z.s', M', T' \rangle} \quad \frac{s[\mu z.s/x], M, T \rightarrow \langle s', M', T' \rangle} {\mu z.s, M, T \rightarrow \langle \mu z.s', M', T' \rangle}
\end{align*}
\]
that the task multiset remains consistent if any of the two sides schedule a new task. Finally, (TR0) and (TR1) specify the concept of recursion. As it can be noticed in the second part of the semantics, most of the semantics rules given for coordination operators only reflect the general semantics of these operators and do not refer to the timing aspect. Note that idling transition is a time pass transition and hence there is no need to define it over the structure of schedules syntax.

Lemma 4.8 For an arbitrary schedule $s$, an arbitrary task set $T$, and a multiset $M$, if $(s, M, T) \rightarrow (s', M', T')$ and $T$ is consistent with respect to $M$, then $T'$ is consistent with respect to $M'$.

Proof. See Appendix B.

The deduction rules for parallel-composition operator are the only rules in which the premises decompose the task multiset in parts. This fact may cause some concerns about consistency of task multiset and its correspondence with the tasks attached to schedule terms. This is not going to cause any problem, since on one hand, time pass and commitment are always possible with the presence of task both in the schedule term and in the task multiset these steps (time pass and commitment) will not cause inconsistency between schedule terms and task multiset. Non the less, time-pass and commitment steps cannot change the consistency of task multiset at all. On the other hand, scheduling new task also adds the new task to both resulting multiset and schedule and all scheduling steps check the consistency of the whole task multiset (not only the decomposed one, in case of parallel-composition semantics).

Despite the similarities between the (un-timed) coordination language and process algebras, like ACP and μCRL the given semantics differs in some cases with the timed-extensions of the mentioned theories [4, 24]. These differences are particularly due to the intentions we had in the design of our timed theory and the decisions we made in the previous section. Usually, real time extensions of process algebras use points of time to represent the time were actions have to take place. However, in our case, timing information is used to estimate the amount of time it takes to perform an action. In our semantics, the absolute time when results of an action (set of actions) are visible depends on the point of time when the actions are scheduled for execution according to the coordination expression or chaotic execution.

The extension of CCS by timing intervals given in [10] is close to our work in that it models timing with intervals that represent relative timing behavior of processes. However, there are two major differences, compared to our approach. First, in [10], the timing information is attached to the syntax of process terms initially, whereas we make a syntactic separation between the two. Second, the semantics given here is more general, because in [10] parallel processes can only be executed at the same time (synchronous parallelism) if the intervals have an intersection. In contrary, we do not have this restriction in our semantics.
If we extend the notion of bisimulation with timing as follows, the previous results of schedule equivalences, i.e., equivalence and congruence (Section 2.4.3), hold for the timed case with the same proofs. A general discussion of different simulation models in timed systems can be found in [21].

**Definition 4.5 (Timed Bisimulation Relation)** We call a relation $R_t$ on timed states a timed-bisimulation if and only if for any two timed states $(s, M, T)$ and $(t, M, T)$ such that $(s, M, T) R_t (t, M, T)$, for all timed states $(s', M'_1, T'_1)$ and $(t', M'_2, T'_2)$:

1. $(s, M, T) \triangleright (s', M'_1, T'_1) \Rightarrow$ for some $(t'', M'', T'')$, $(t', M', T') R_t (t'', M'', T'')$
2. $(t, M, T) \triangleright (t', M'_2, T'_2) \Rightarrow$ for some $(s'', M'', T'')$, $(s', M', T') R_t (t'', M'', T'')$
3. $(s, M, T) \triangleright (t, M, T)$
4. $(t, M, T) \triangleright (s, M, T)$

Two schedules $s$ and $t$ are timed-bisimilar (written: $s \cong_t t$) if and only if there exists a timed-bisimulation relation $R_t$ such that for all multisets $M$, $(s, M, \square) R_t (t, M, \square)$.

As in Chapter 2, we re-use the bisimulation relation between two schedules (defined above) for a schedule and a GAMMA program and two GAMMA programs. Due to the natural extension of the semantics that separates the functionality, behavior, and timing parts (resulting in a similar semantics for schedule operators), all schedule equalities in Proposition 2.17 remain sound with little or no change in the proofs. Those proofs that need a little change are given in Appendix B.

Furthermore, the notion of the Most General Schedule from the untimed setting remains the same in the timed setting. However, since the tasks are attached to schedule terms, we have to play a bit with the previously given proof of bisimulation. The new proof is also given in Appendix B.

**Theorem 4.9 (Timed Most General Schedule)** For all timed-GAMMA programs $R$, $MGS(R) \cong_t R$.

**Proof.** See Appendix B. ☐

**Example 4.3 (Reactor shutdown system)** This example is a simplified version of an industrial application used for nuclear reactor control, studied in [22]. The shutdown system we are modelling checks the pressure and the power of a reactor and if it finds them exceeding the allowed safe values ($PrLimit$ and $PwLimit$), it starts a timer. Checking the conditions takes a time between
As the timer starts, the normal systems for controlling the reactor may cool it down to reach the safe area. But, if after the timer sends the timeout signal (in $t_{	ext{tick}}$ time with a drift) the system finds the values still above the safety thresholds, it generates a shutdown command to stop the whole reactor, and allows it to cool down. The reactor will be allowed to turn on again, only after $t_{	ext{coolDown}}$ time passed from the time both measures reach their safe value.

ShutdownSystem =

\[
\begin{aligned}
\text{checkDanger} &= \text{power}?, \text{pressure}?: \mapsto \left\{ \begin{array}{ll}
\text{pressure} > \text{PrLimit} \land \text{power} > \text{PwLimit}, \\
\text{timer} &= \mapsto \text{tick}, \\
\text{shutCheck} &= \text{power}?, \text{pressure}?: \text{tick} \mapsto \\
\text{pressure} > \text{PrLimit} \land \text{power} > \text{PwLimit}, \\
\text{shutdown} &= \mapsto \text{ShutdownCmd}, \\
\text{restart} &= \text{power}?, \text{pressure}? : \text{ShutdownCmd} \mapsto \\
\text{pressure} &\leq \text{PrLimit} \lor \text{power} \leq \text{PwLimit}
\end{array} \right.
\end{aligned}
\]

\[T_{\text{checkDanger}} = T_{\text{shutCheck}} = [t_{\text{minAnd}}, t_{\text{maxAnd}}] \]
\[T_{\text{timer}} = [t_{\text{tick}} - \text{drift}, t_{\text{ick}} + \text{drift}] \]
\[T_{\text{shutdown}} = [t_{\text{minShutdown}}, t_{\text{maxShutdown}}] \]
\[T_{\text{restart}} = [t_{\text{coolDown}}, t_{\text{coolDown}}] \]

The initial multiset of the above GAMMA program consists of tuples representing current pressure and power. The identifiers power and pressure abbreviate pairs of name and value. The program consists of rules for checking pressure and power values in normal (checkDanger), the timer (timer), the power and pressure check after the timer generated the timeout tick (shutCheck), generating shutdown command (shutdown) and restarting the system when the conditions return to normal mode (restart).

The chaotic behavior of the timed-GAMMA program cannot provide any guarantee for the shutdown system to work in time, but the following simple schedule may help in achieving a reliable upper-bound for shutting down the system:

\[\text{reactorControl} = \mu X. \text{checkDanger} \rightarrow \]
\[\text{timer} \parallel \text{shutCheck} \rightarrow \]
\[(\text{shutdown} ; \text{reactorRestart})[X][X] \]

\[\text{reactorRestart} = \mu Y. \text{restart} \rightarrow \text{reactorControl}[Y] \]

The above schedule first checks the pressure and power values. If it finds them dangerous, sets the timer and waits for it to timeout (shutCheck requires the tick to proceed). After the timeout occurs, if the values are still dangerous, shutdown and restart commands are generated and systems waits for normal condition to restart, otherwise system restarts executing its control schedule.
4.4 Urgency Operator

For some coordination operators (not yet included in current schedules syntax) it may be needed to differentiate between idling and normal time pass transitions that are spent on computation. This can be done by comparing task multisets at the two sides of the transition (which remains the same in case of idling). A particular example of such an operator is the urgency operator, denoted by $\nu$ following [29], which forces the schedule not to idle:

\[
\begin{align*}
(U0) & \quad \langle s, M, T \rangle \xrightarrow{\nu} \langle s', M', T' \rangle \quad T \neq T' \quad \langle \nu(s), M, T \rangle \xrightarrow{\nu} \langle \nu(s'), M', T' \rangle \\
(U1) & \quad \langle s, M, T \rangle \xrightarrow{\nu} \langle s', M', T' \rangle \\
(U2) & \quad \langle \nu(s), M, T \rangle \xrightarrow{\nu}
\end{align*}
\]

To put it informally, $\nu$ filters idle transitions and does not allow them to happen when applied to coordination expressions. A deadline urgency operator ($\nu_I$) can be defined to force the sum of idling transition to be within an interval:

\[
\begin{align*}
(IU0) & \quad \langle s, M, T \rangle \xrightarrow{\nu_I} \langle s', M, T' \rangle \quad T \neq T' \quad \langle \nu_I(s), M, T \rangle \xrightarrow{\nu_I} \langle \nu_I(s'), M', T' \rangle \\
(IU1) & \quad \langle s, M, T \rangle \xrightarrow{\nu_I} \langle s', M', T' \rangle \\
(IU2) & \quad \langle s, M, T \rangle \xrightarrow{\nu_I} \langle s', M, T' \rangle \quad T = T' \quad t < I \\
& \quad \langle \nu_I(s), M, T \rangle \xrightarrow{\nu_I} \langle \nu_I-(s'), M, T' \rangle \\
(IU2) & \quad \langle s, M, T \rangle \xrightarrow{\nu_I} \langle s', M, T' \rangle \quad T = T' \quad t \in I \\
& \quad \langle \nu_I(s), M, T \rangle \xrightarrow{\nu_I} \langle \nu(s'), M, T' \rangle \\
(IU3) & \quad \langle \nu_I(s), M, T \rangle \xrightarrow{\nu_I}
\end{align*}
\]

4.5 Reasoning

Because they abstract from real time, classic Temporal Logics are only good for qualitative reasoning about time. Several extension have been proposed for different Temporal Logics to support specification of real-time properties. Without going deep into the comparison of these extensions, we use the one that comes in line with our extension of GAMMA and coordination with intervals, namely
Metric Temporal Logic (MTL, or also Bounded-Operator Temporal Logic). The interested reader can find detailed comparisons of expressive power and verification complexity between temporal logics in [3, 17, 7]. (The results of these comparisons also show that our choice of MTL is possibly the best for our context.) Since instantaneous actions may happen in our semantics, in addition to the ordering imposed by time, we need an ordering imposed by causality in order to represent and order the states caused by such an action. Hence, we use a richer notion of time (containing both real and causal time) than in Metric Temporal Logic, and hence we call the slightly modified temporal logic (that takes also the causal ordering of events into account) Metric Rich Temporal Logic (MRTL).

The syntax of MRTL is given in Figure 4.8. In this syntax, temporal operators are bounded with an interval. The until operator \( U \) means that at some time within interval \( I \) the second formula holds, before which the first formula keeps being true (from now). As before, other interesting temporal operators can be constructed from the presented basic ones. For example, \( F \) is defined as true \( U F \), and means that \( \varphi \) will hold some time during the interval \( I \), or \( G \) is defined as \( \neg F \) and means that \( \varphi \) has to be true during the whole interval \( I \). The next operator is not included in MRTL because it has no intuitive meaning in timed settings (or the same intuitive meaning as \( F \)). To define the semantics of MRTL over timed schedules we need to define timed computations of such specifications. To verify GAMMA programs one may use the same definitions or simply verify the corresponding most general schedule.

**Definition 4.6 (Timed-Closure-Transition)** We define that the state \((s, M, T)\) can perform the timed-closure-transition \( (s, M, T) \xrightarrow{t} (s', M', T') \) if and only if there exist transitions with the labels \( t_0 \ldots t_n \), such that \( (s, M, T) \xrightarrow{t_0} (s_0, M_0, T_0) \xrightarrow{t_1} \ldots \xrightarrow{t_n} (s', M', T') \) and \( \sum_{i=0}^{n} t_i = t \).

**Definition 4.7 (Timed-Computation)** The transition \( (s, M, T) \xrightarrow{\sigma t} (s', M', T') \) is a timed-computation of \((s, M, T)\) if and only if \( (s, M, T) \xrightarrow{t} (s', M', T') \) and \( (s', M', T') \xrightarrow{\sigma t} (s'', M'', T'') \).

**Definition 4.8 (Timed-Run (timed-state-sequence))** The tuple \((\alpha, \tau)\) where \( \alpha \) is a sequence of states and \( \tau \) is a sequence of absolute transition times, is a timed-run of the schedule \( s \) and the multiset \( M \) if and only if it has the following:

\[
\begin{align*}
\alpha &= s_0, s_1, \ldots, s_n, \\
\tau &= t_0, t_1, \ldots, t_n
\end{align*}
\]
(α, τ), t, i \models p \iff |α| \geq i, \tau(i) \leq t \leq \tau(i + 1), \text{ and } M_i \models p \\
(\text{for atomic formula } p)

(α, τ), t, i \models \neg \varphi \iff |α| \geq i, \tau(i) \leq t \leq \tau(i + 1), \text{ and } (α, τ), i, i \not\models \varphi

(α, τ), t, i \models \varphi_1 \lor \varphi_2 \iff (α, τ), t, i \models \varphi_1 \text{ or } (α, τ), t, i \models \varphi_2

(α, τ), t, i \models \varphi_1 \cup \varphi_2 \iff \text{there exist } t' \text{ and } j \text{ such that } t' \in t + I \text{ and } (α, τ), t', j \models \varphi_2 \text{ and for all } k \text{ and } t'' \text{ such that } i \leq k < j \text{ and } \tau(k) \leq t'' \leq \tau(k + 1): \\
(α, τ), t'', k \models \varphi_1

---

**Figure 4.9: Semantics of Metric Rich Temporal Logic Formulas**

Properties \(α(i)\) and \(τ(i)\) are used to denote the \(i\)th element of the sequences \(α\) and \(τ\). If the lengths of the sequences are less than \(i\) then \(α(i)\) is undefined and \(τ(i) = \infty\):

1. Initiality: \(α(0) = (s, M, \emptyset)\),

2. Sequence: \(\forall i \in \mathbb{N} \quad 0 \leq i < |α|, \exists t \in \text{Time} \cdot s \quad α(i) \rightarrow^t α(i + 1) \text{ and } τ(i + 1) = τ(i) + t\).

**Definition 4.9 (Timed-Behavior)** The set of all timed-runs of a schedule \(s\) with initial state \(M\) is called the timed-behavior of the schedule and is denoted by \(\prod(s, M)\).

The behavior of a schedule is closed under shifting the time origin and scaling the time unit [18]. It means that if \((α, τ)\) is in \(\prod(s, M)\), then \((α, a * τ + c)\) is in \(\prod(a * s, M)\), where, \(a * τ + c\) is a new function mapping each state \(α(i)\) to \(a * τ(i) + c\) and \(a * s\) is a new specification with all \(τ.I\) being replaced by \(a * (τ.I)\). In other words, the given semantics (and, in general, Timed-Computations) do not refer to absolute time (as it appears from in the semantic rules like (TimePass)), and without loss of generality the time origin can be assumed 0. So, from now on we assume that for all timed-runs \((α, τ)\) in the behavior of a program (or a schedule) \(τ(0) = 0\). As we mentioned before, to verify properties of schedules, we implicitly rule out idling transitions. This can be formally done by applying the urgency operator to all rules participating in the schedule (apart from those appearing as conditions in rule conditional).

The formal semantics of MRTL is described in Figure 4.9, based on the definition of a timed-run. The formula \(\varphi\) holds for the timed-run \(θ\) if and only if \(θ, 0, 0 \models \varphi\). The schedule \(s\) with the initial multiset \(M\) satisfies \(\varphi\) if and only if for all timed-computations \(θ \in \prod(s, M)\) it holds that \(θ, 0, 0 \models \varphi\).
Example 4.4 (Dying dining philosophers, Timed correctness properties)

Our real-time version of the dining philosophers problem (Example 4.1) should have the following properties:

1. No two neighboring philosophers may eat (i.e., share a specific fork) at the same arbitrary time (same as before).

2. No philosopher dies of hunger, in other words, when s/he is hungry, s/he will receive two forks in a given amount of time (at most \( t_{\text{die}} \)).

The following statements formalize these requirements in MRTL:

- The mutual exclusion property in Example 2.9 is extended naturally to the following property:

\[
\text{G}_{[0,\infty)}(((PE, i) \in \text{room}) \Rightarrow (PE, (i - 1) \mod n) \not\in \text{room} \\
\land \\
(PE, (i + 1) \mod n) \not\in \text{room})
\]

for all \( i: 0 \leq i < n \).

We may turn this invariant property to a more interesting timed property by stating that: Every philosopher desires a kind of discipline that his/her fork remains for him/her for a specified amount of time after s/he has finished eating:

\[
\text{G}_{[0,\infty)}(((PE, i) \in \text{room}) \Rightarrow \text{G}_{[0, t_{\text{die}}]} ((PE,(i - 1) \mod n) \not\in \text{room} \\
\land \\
(PE, (i + 1) \mod n) \not\in \text{room})
\]

- \( \text{G}_{[0,\infty)}(((PS, i) \in \text{room}) \Rightarrow \text{F}_{[0, t_{\text{die}}]}((PE, i) \in \text{room})) \)

This property asserts that for all states, if a philosopher gets hungry and sits in his/her chair, waiting for forks, s/he will receive them eventually before the deadline \( t_{\text{die}} \) after which the philosopher dies of hunger (this is actually the reason why the problem is named dying dining philosophers).

As we have seen in Example 2.9, the sequential and parallel schedules given for the dining philosophers example both guarantee that a hungry philosopher will find the opportunity to eat in the future. But, in un-timed settings we cannot compare the time needed for serving a hungry philosopher in sequential and parallel schedules. Now, in the timed setting however, one can observe that in the sequential schedule, it takes at least \( t_{\text{think}} + n \times t_{\text{eat}} \) for a philosopher to be served. On the other hand, for the parallel schedule there exists a run in which this time is reduced to \( t_{\text{think}} + 2 \times t_{\text{eat}} \) (and to \( t_{\text{think}} + t_{\text{eat}} \) if the number of philosophers is odd).

The result for the parallel schedule is due to the semantics of parallel composition. It allows independent rules to be executed together, with taking the
maximum estimation of execution time between them. In the case when the number of philosophers is odd, members of the two subsets of philosophers (odd numbered and even-numbered ones) can eat concurrently, and hence, reduce the service time in this particular run to $t_{eat}$. If the number of philosophers is even, there will be a conflict between philosophers number 0 and $n-1$, and hence one of them should wait one more $t_{eat}$ in addition to the above estimation.

Example 4.5 (The elevator system, Timed correctness properties) The un-timed property of the elevator system (in Example 2.10) is extended such that any request from inside or outside should be service within $t_{service}$ time:

$$G_{[0,\infty]} (((\text{extStop}, i), \text{on}) \in M \Rightarrow F_{[0,t_{service}]} (((\text{extStop}, i), \text{off}) \in M)$$

$$\wedge (((\text{inStop}, i), \text{on}) \in M \Rightarrow F_{[0,t_{service}]} (((\text{inStop}, i), \text{off}) \in M)$$

Again, the general timed-GAMMA program (given in Example 11) cannot satisfy this property with any arbitrarily large $t_{service}$ (following the same reasoning as for the un-timed case). However, the given schedule in Example 7 services every request after visiting at most $n - 1$ other floors two times. Hence, by taking a large enough $t_{service}$, for example, $t_{service} \geq 2 \times n \times (\text{StepTime} + 2 \times \text{StartTime} + 2 \times \text{MaxServiceTime})$, the above property holds.

Example 4.6 (The reactor shutdown system, Correctness properties) Suppose that we specify our correctness property as follows: If both inputs of power and pressure exceed their limit and remain high for at least $t_{stable}$ time, then the shutdown signal will be generated by the specified system in less than $t_{reaction}$ time:

$$ShutdownProp = G_{[0,\infty)} \text{ (DangerousCondition)}$$
$$\quad \Rightarrow F_{[0,t_{reaction}]} (\text{ShutdownCmd} \in M))$$

$$\text{DangerousCondition} = G_{[0,t_{stable}]} \text{ (pressure} > \text{PrLimit} \wedge \text{power} > \text{PuLimit}).$$

Obviously, the general GAMMA program does not satisfy the above property. What we presented as a coordination in Example 12 is an attempt to satisfy the above property and as a result the property holds if both inputs remain high for a minimum required time: $t_{stable} \geq t_{maxAnd} + t_{kick} + \text{drift}$ then, the system will shutdown in $t_{reaction} \leq t_{maxShutdown}$.

As shown in the examples throughout this paper, the given coordination expressions coincide with the properties desired from the system behavior. This fact invites us to investigate the way schedules can be derived from correctness properties. Similar to Most General Schedules, we propose studying the Most General Correct Schedule with respect to temporal properties in both timed and un-timed cases.
Chapter 5

Conclusion and Research Directions

We presented a model for system design which deals with different aspects of systems as separate orthogonal concerns. A very basic subset of the GAMMA language is used to define functionality of components, a coordination (scheduling) language for exploiting functionalities in a special order, and a timing model for representing estimations of execution time. Linear-time temporal logic and one of its real-time extensions are used to reason about designed systems. This orthogonality means that specifications of each aspect (functionality, coordination and timing) can be re-used when changing other aspects or even in the absence of some other aspects. Figure 5.1 depicts a schematic view of our method.

We plan to extend our approach in the following steps:

1. Extracting and refining schedules (based on properties): As shown in the last section, some schedules conform to some correctness properties specified by temporal logic formulas. Finding the Most General notion of such Correct Schedules seems to be a challenging problem.

2. Structuring the multiset in GAMMA: We are planning to provide some basic structuring and typing mechanism for our basic GAMMA language in order to make it useable for industrial case studies. A current extension of GAMMA in this direction, namely Structured-GAMMA [14], provides a starting point.

3. Automatic reasoning about GAMMA programs and schedules: We will attempt to translate a coordinated program to the input languages of automatic verification tools. This will mechanize verification of schedule correctness.

4. Extension by the distribution aspect: Studying the effects of distribution of data and program parts on timing properties and defining distribution
policies in GAMMA is another interesting topic for our future research and a step toward (distributed) implementation.

5. Studying fault tolerance and reliability: Replication can provide a basis for improving fault tolerance and reliability of a system. The GAMMA model represented in this report is indifferent to replication of single rules or replication of one rule with several names. This feature can provide a basis for studying process replication in our design model. Data replication, on the other hand, is also another interesting topic in this paradigm.

6. A detailed timing model: The timing model used here could be extended by presenting more detailed pieces of timed-functionality (e.g., timing for communication, for pattern matching in the multiset, and for performing computation). This extension may bring timing information to scheduling phase (instead of the current assumption of zero time for scheduling) and termination, too. Also, this detailed information about timing may involve timing with other aspects like distribution.

7. Specification of real-world case studies: Capabilities and shortcomings of our design method can be further investigated only by specification and design of more complex real-world examples. Furthermore, design of real-world examples may help us with inspiring possible ways of transformation to the implementation domain.
Acknowledgments

The authors would like to thank colleagues at the University of California at Irvine namely, Sandeep Shukla, Angelo Corsaro, Rajesh Gupta, and Douglas Schmidt, for their constructive discussion and suggestions on GAMMA and timed-GAMMA. Also, we appreciate Mark Geilen’s and Peter Cuijpers’ insightful comments on early drafts of this report. We express our appreciations for the kind effort of Jan Friso Groote and Judi Romijn in reviewing the final version of this report.
Bibliography


Appendix A

Proofs of Untimed Properties

In this appendix we are mainly aiming at proving the following properties:

1. Bisimilarity of the schedules in Proposition 2.17: This is done by defining a relation and proving it to be a bisimulation relation according to Definition 2.15.

2. Bisimilarity of GAMMA programs and their corresponding most general schedule: This is done based on some results for a single rule program \( \{r\} \) and its single most general schedule \( \mu X.r \triangleleft (r \parallel X) \):
   (a) First, we prove that a single most general schedule is bisimilar with the rule in parallel with the single most general schedule (Lemma A.2).
   (b) Then, we prove that if a single most general schedule performs a transition, then the resulting schedule is bisimilar to the result of the transition (Lemma A.4).
   (c) Subsequently, it is shown that the schedule comprising of two single most general schedules in parallel is bisimilar to the single most general schedule (Lemma A.6).
   (d) Using the above lemmas, we are able to prove the equivalence of a single rule program and its single most general schedule in Lemma A.7.

After all, we lift the result for a single rule program to an arbitrary program in Theorem 2.18 by an induction on the size of the program.

3. The last result in this appendix is the fact that union of programs corresponds to parallel composition of schedules (Proposition 2.19).
Lemma 2.10 (Section 2.3.3) If a program $R$ can perform an arbitrary $\sigma$ transition $(R, M) \xrightarrow{\sigma} (R, M')$, then for an arbitrary rule $r$, $(R \cup \{r\}, M) \xrightarrow{\sigma} (R \cup \{r\}, M')$. (i.e., if a program can perform a computation, a bigger program can perform it, too.)

Proof. By an induction on the proof depth for the $\sigma$ transition. For simplicity, we consider the depth of the proof up to the semantics of a single rule:

1. Induction base: If $(R, M) \xrightarrow{\sigma} (R, M')$ has a proof of depth one, then this transition is due to (ProgComp) (proofs that use (ProgPar) need at least one more step to prove the premises). It follows from (ProgComp) that there is an $r' \in R$ such that $(r', M) \xrightarrow{\sigma} (r', M')$. Because $r' \in R \cup \{r\}$, we can apply (ProgComp) again using this transition and conclude that $(R \cup \{r\}, M) \xrightarrow{\sigma} (R \cup \{r\}, M')$.

2. If $(R, M) \xrightarrow{\sigma} (R, M')$ by a proof of depth $n$ (greater than one), then this transition is due to (ProgPar) and there exist two substitutions $\sigma_1$ and $\sigma_2$ such that $(R, M) \xrightarrow{\sigma_1} (R, M_1)$, $(R, M) \xrightarrow{\sigma_2} (R, M_2)$, $\sigma = \sigma_1 \circ \sigma_2$, and $M' = M[\sigma_1, \sigma_2]$. According to the induction hypothesis (because the premises should have a proof of depth $n - 1$) $(R \cup \{r\}, M) \xrightarrow{\sigma} (R \cup \{r\}, M_1)$, $(R \cup \{r\}, M) \xrightarrow{\sigma} (R \cup \{r\}, M_2)$, and $M' = M[\sigma_1, \sigma_2]$. It follows from (ProgPar) that $(R \cup \{r\}, M, \sigma_1 \circ \sigma_2)$ and $M' = M[\sigma_1, \sigma_2]$. It follows from (ProgPar) that $(R \cup \{r\}, M, \sigma_1 \circ \sigma_2)$. □

Corollary 2.11 (Section 2.3.3) For arbitrary GAMMA programs $R_1$ and $R_2$, if $(R_1, M) \xrightarrow{\sigma} (R_1, M')$, then $(R_1 \cup R_2, M) \xrightarrow{\sigma} (R_1 \cup R_2, M')$.

Proof. It follows from Lemma 2.10, using an induction on the size of $R_2$. □

Lemma 2.12 (Section 2.3.3) $(R, M) \xrightarrow{\sigma} (R, M[\sigma])$ if and only if for all rules $r \in R$, one of the following cases holds:

1. $(R \setminus \{r\}, M) \xrightarrow{\sigma} (R \setminus \{r\}, M[\sigma])$

2. $(R \setminus \{r\}, M) \xrightarrow{\sigma} (R \setminus \{r\}, \{r\}, M[\sigma_1]), (\{r\}, M) \xrightarrow{\sigma} (\{r\}, M[\sigma_2]), \sigma = \sigma_1 \circ \sigma_2$ and $M' = M[\sigma_1, \sigma_2]$

3. $(\{r\}, M) \xrightarrow{\sigma} (\{r\}, M[\sigma])$

Proof. $(1) \Rightarrow$: By an induction on the proof depth for the $\sigma$ transition:

1. Induction base: If $(R, M) \xrightarrow{\sigma} (R, M[\sigma])$ with a proof of depth 1, then this transition can be only due to (ProgComp) and there is a rule $r' \in R$ such that $(r', M) \xrightarrow{\sigma} (r', M[\sigma])$. If $r$ is the same as $r'$, then according to
(ProgComp), \( \langle \{ r \}, M \rangle \stackrel{\sigma}{\rightarrow} \langle \{ r \}, M[\sigma] \rangle \) (item 3). If they are different, then since \( r' \in R \setminus \{ r \} \) it follows from (ProgComp) that \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma] \rangle \) (item 1).

2. Induction step: Suppose that \( \langle R, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R, M[\sigma] \rangle \) with the proof of depth \( n \) (greater than one), then the transition can be only due to (ProgPar) which means that \( \langle R, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R, M[\sigma_1] \rangle \), \( \langle R, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle R, M[\sigma_2] \rangle \), \( \sigma = \sigma_1, \sigma_2 \) and \( M \models \sigma \equiv \sigma_1 \cong \sigma_2 \). Both transitions of \( \sigma_1 \) and \( \sigma_2 \) have a proof depth of less than \( n \), and thus the induction hypothesis applies to both. Considering an arbitrary rule \( r \in R \), applying the induction hypothesis leads to 9 different cases. But due to symmetry, examining the following 6 cases completes the proof:

(a) \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1] \rangle \) (item 1) and \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_2] \rangle \) (item 1). Then it follows from (ProgPar) that \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1 \cong \sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1, \sigma_2] \rangle \) and this completes the proof (item 1).

(b) \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1] \rangle \) (item 1), \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_2] \rangle \), \( \sigma_1 = \sigma_{21, 22} \) and \( M \models \sigma_1 \equiv \sigma_2 \) (item 2). Then we compose the two transitions of \( \sigma_1 \) and \( \sigma_2 \) using (ProgPar) (according to Proposition 2.7 \( M \models \sigma_1 \equiv \sigma_2 \)). It follows that \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1, \sigma_2] \rangle \) and \( \langle \{ r \}, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle \{ r \}, M[\sigma_2] \rangle \) (item 2, independence condition holds following Theorem 2.9).

(c) \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1] \rangle \) (item 1) and \( \langle \{ r \}, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle \{ r \}, M[\sigma_2] \rangle \) (item 3). Then the lemma trivially holds (item 2).

(d) \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1] \rangle \), \( \langle \{ r \}, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle \{ r \}, M[\sigma_2] \rangle \) (item 2), \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_2] \rangle \), \( \langle \{ r \}, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle \{ r \}, M[\sigma_1] \rangle \) (item 2), \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_1 \equiv \sigma_2] \rangle \) (item 2), \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle R \setminus \{ r \}, M[\sigma_2 \equiv \sigma_1] \rangle \) (item 2). Then according to (ProgPar) and Proposition 2.7, \( \langle \{ r \}, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle \{ r \}, M[\sigma_1 \equiv \sigma_2] \rangle \) and \( \langle R \setminus \{ r \}, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle \{ r \}, M[\sigma_2 \equiv \sigma_1] \rangle \) and since composition of substitutions is associative and commutative by definition, \( \sigma = \sigma_{21, 22} \) (item 2).

(f) \( \langle \{ r \}, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle \{ r \}, M[\sigma_1] \rangle \) (item 3), \( \langle \{ r \}, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle \{ r \}, M[\sigma_2] \rangle \) (item 3). Composing \( \sigma_1 \) and \( \sigma_2 \) (by (ProgPar)) results in item 3.

(2) \( \Leftarrow \):

1. Immediate consequence of Lemma 2.10.

2. According to Corollary 2.11 \( \langle R, M \rangle \stackrel{\sigma_1}{\rightarrow} \langle R, M[\sigma_1] \rangle \) and \( \langle R, M \rangle \stackrel{\sigma_2}{\rightarrow} \langle R, M[\sigma_2] \rangle \). It follows from (ProgPar) that \( \langle R, M \rangle \stackrel{\sigma_1 \equiv \sigma_2}{\rightarrow} \langle R, M[\sigma_1 \equiv \sigma_2] \rangle \).
3. It follows from Corollary 2.11 that \( \langle R, M \rangle \xrightarrow{\alpha} \langle R, M[\alpha] \rangle \).

We will use the notation \( \equiv \) to denote syntactic equivalence of two schedules. Two syntactically equivalent schedules are (according to Theorem 2.15) always bisimilar. The smallest bisimulation relation that relates all syntactically equivalent schedules is denoted by \( id \).

**Theorem 2.16 Congruence:** Bisimilarity is a congruence with respect to all schedule operators.

**Proof.** To prove the theorem we should prove the following propositions: For all schedules \( s, s', t, \) and \( t' \) and arbitrary rules \( r \) and \( r' \), if \( r \equiv r', t \equiv t' \) and \( s \equiv s' \) then :

1. \( r \rhd t \equiv r' \rhd t' \]
2. \( s || t \equiv s' || t' \]
3. \( s \rhd t \equiv s' \rhd t' \]
4. \( s \equiv s' \)

1. If \( r \equiv r', t \equiv t' \) and \( s \equiv s' \), we denote the minimum bisimulation relation relating \( r \) and \( r' \) as \( R_r \), \( t \) and \( t' \) as \( R_t \), and \( s \) and \( s' \) as \( R_s \), we construct the new relation \( R \cong R_r \cup R_t \cup R_s \). Then:

1. If \( \langle r \rhd s[t], M \rangle \cong \langle u, M' \rangle \), then:
   - Either \( \langle r, M \rangle \cong \langle u, M' \rangle \) then according to \( RC_1 \), we should have \( \langle t, M \rangle \cong \langle u', M' \rangle \). But since \( r \equiv r' \) and \( t \equiv t' \), \( \langle r', M \rangle \cong \langle u', M' \rangle \) and \( \langle s, lVI \rangle \cong \langle u', M' \rangle \) is in \( R_2 \) and hence, in \( R \). It follows from \( RC_1 \) that \( \langle r' \rhd s'[t'], M \rangle \cong \langle u', M' \rangle \) and since \( \langle u, M' \rangle \cong \langle u', M' \rangle \) is in \( R \), it completes the proof of this case.
   - Or \( \langle r, M \rangle \cong \langle u, M' \rangle \). Then according to \( RC_0 \) we should also have: \( \langle s, M \rangle \cong \langle u, M' \rangle \) since \( r \equiv r' \) and \( s \equiv s' \), \( \langle r, M \rangle \) (an immediate consequence of Definition 2.15), \( \langle s', M \rangle \cong \langle u', M' \rangle \). It follows from \( RC_0 \) that \( \langle r' \rhd s'[t'], M \rangle \cong \langle u', M' \rangle \).

2. If \( \langle r' \rhd s'[t'], M \rangle \cong \langle u, M' \rangle \), then following the same reasoning of the previous item \( \langle r \rhd s[t], M \rangle \cong \langle u', M' \rangle \) and \( \langle u, M \rangle \cong \langle u', M' \rangle \) in \( R \).

3. If \( \langle r \rhd s[t], M \rangle \) then according to \( RC_2 \) or \( RC_3 \) we should have either \( \langle r, M \rangle \cong \langle s, M \rangle \) or \( \langle r, M \rangle \cong \langle t, M \rangle \). But since \( r \equiv r', t \equiv t' \) and \( s \equiv s' \), either \( \langle r', M \rangle \cong \langle s', M \rangle \) or \( \langle r', M \rangle \cong \langle t', M \rangle \). So in both cases, we have \( \langle r', M \rangle \cong \langle s'[t'], M \rangle \).
4. Symmetric to item 3.

2. As in the previous items, we construct the relation \( R \) by \( R \overset{\Delta}{=} R_1 \cup R_2 \cup \{(s ; t, M), (s' ; t', M')\} | ((s, M), (s', M)) \in R_1 \wedge ((t, M), (t', M)) \in R_2 \}. Then:

1. If \( (s ; t, M) \overset{\Delta}{=} (u, M') \). Then this transition is due to either of the following rules:
   
   (a) (SO): Then, \( (s, M) \overset{\Delta}{=} (u', M') \) and \( u \equiv u' \wedge t' \). But since \( s \equiv s' \), \( (s', M) \overset{\Delta}{=} (u'', M') \) and \( ((u', M'), (u'', M')) \in R_1 \). It follows from (S0) that \( (s' ; t', M) \overset{\Delta}{=} (u'', t', M') \) and \( ((u', t', M'), (u'', t', M')) \in R \).

   (b) (SI): Then, \( (s, M) \overset{\Delta}{=} (u, M') \) and \( t \equiv t' \). Since \( s \equiv s' \) and \( t \equiv t' \), \( (s', M) \overset{\Delta}{=} (u', M') \) and \( ((u', M'), (u', M')) \in R_1 \). Then, it follows from (S1) that \( (s' ; t', M) \overset{\Delta}{=} (u', t', M') \) and \( ((u, t, M'), (u', t', M')) \in R \).

2. If \( (s ; t, M) \overset{\Delta}{=} (u, M') \), then following the same reasoning as in previous item \( (s' ; t', M) \overset{\Delta}{=} (u', M') \) and \( ((u', M'), (u', M')) \in R \).

3. \( (s ; t, M) \overset{\Delta}{=} \) if and only if \( (s, M) \overset{\Delta}{=} (u, M') \) and \( (t, M) \overset{\Delta}{=} (u', M') \), and \( s \equiv s' \) and \( t \equiv t' \) this holds if and only if \( (s', M) \overset{\Delta}{=} (u', M') \) and \( (t', M') \overset{\Delta}{=} (u', M') \), thus according to (S2) \( (s' ; t', M) \overset{\Delta}{=} (u', M') \).

3. Assume the relation \( R \) to be \( R \overset{\Delta}{=} R_1 \cup R_2 \cup \{(s ; t, M), (s' ; t', M')\} | ((s, M), (s', M')) \in R_1 \wedge ((t, M), (t', M')) \in R_2 \), for all multisets \( M \), where \( R_1 \) and \( R_2 \) have their previous definition. The proof of bisimilarity is similar to that of 2.

4. It trivially holds; since we only consider closed terms (i.e., \( s \) and \( s' \) are bisimilar) introducing recursion does not influence bisimilarity.

\( \square \)

For the particular schedule \( s \) (similarly the program), the multiset \( M \), and the substitution \( \sigma \), if there is no schedule \( s' \) and multiset \( M' \) such that \( (s, M) \overset{\Delta}{=} \) is a valid transition, then we write \( (s, M) \overset{\Delta}{=} \). It follows trivially from the given semantics that \( (\forall \sigma; (s, M) \overset{\Delta}{=} \Rightarrow (s, M) \overset{\Delta}{=} \).

**Corollary A.1** For two arbitrary schedules \( s \) and \( s' \) and multiset \( M \), if \( s \equiv s' \) then \( (s, M) \overset{\Delta}{=} \iff (s', M) \overset{\Delta}{=} \).

**Proof.** Straightforward consequence of the Definition 2.15. \( \square \)

**Proposition 2.17** Some bisimilar schedules: According to the given semantics, the following bisimilarities hold, for all schedules \( s, s_1, s_2, s_3 \) and rule \( r \):

74
1. \( \text{skip} \cdot s \equiv s \)
2. \( s \cdot \text{skip} \equiv s \)
3. \( s_1 \cdot (s_2 \cdot s_3) \equiv (s_1 \cdot s_2) \cdot s_3 \)
4. \( \text{skip} || s \equiv s \)
5. \( s_1 || s_2 \equiv s_2 || s_1 \)
6. \( (s_1 || s_2) || s_3 \equiv (s_1 || s_2) \cdot s_3 \)
7. \( r \cdot \text{skip} \equiv \text{skip} \)
8. \( r \cdot r \equiv r \cdot \text{skip} \equiv r \)
9. \( \text{skip} \cdot \text{skip} \equiv \text{skip} \cdot \text{skip} \)
10. If \( y \) is not present in \( s \), \( \mu x. s \equiv \mu y. s[y/x] \). Where, \( s[y/x] \) is the syntactic substitution of recursion variable \( x \) with \( y \).

**Proof.**

1. For all \( M \) and \( \sigma \), assume the bisimulation relation \( R \overset{=}{=} \text{id} \cup \{(\text{skip} \cdot s, M), (s, M)\} \), for all schedules \( s \) and all multisets \( M \). For the \( \text{id} \) subset of \( R \) it is obvious that it is a bisimulation according to Theorem 2.15. (We do not repeat reasoning of this part in the future.) For the rest, we have:

   (1) \(\langle \text{skip} \cdot s, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \Rightarrow \text{ either } \langle \text{skip}, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \overset{(S0)}{=} \langle t, M' \rangle \land \langle \text{skip}, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \overset{(S1)}{=} \langle t, M' \rangle \)

   (2) \(\langle s, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \Rightarrow \langle s, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \land \langle \text{skip}, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \overset{(S1)}{=} \langle t, M' \rangle \)

   (3), (4) \(\langle s, M \rangle \overset{\sigma}{\to} \langle \text{skip}, M \rangle \lor \langle s, M \rangle \overset{\sigma}{\to} \langle \text{skip} \cdot s, M \rangle \)

2. For all \( M \) and \( \sigma \), assume the bisimulation relation \( R \overset{=}{=} \text{id} \cup \{(\text{skip} \cdot s, M), (s, M)\} \), for all schedules \( s \) and all multisets \( M \):

   (1) \(\langle s \cdot \text{skip} \cdot M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \Rightarrow \text{ either } \langle s, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \overset{(S0)}{=} \langle t, M' \rangle \land \text{ (S1) } \text{ or } \langle \text{skip}, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \)

   (2) \(\langle s, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \Rightarrow \langle s, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \land \langle \text{skip}, M \rangle \overset{\sigma}{\to} \langle t, M' \rangle \)

   (3), (4) \(\langle s, M \rangle \overset{\sigma}{\to} \langle \text{skip}, M \rangle \lor \langle s, M \rangle \overset{\sigma}{\to} \langle \text{skip} \cdot s, M \rangle \)

3. We define the relation \( R \overset{=}{=} \text{id} \cup \{(s ; t ; u, M), (s ; t ; u, M)\} \), for all schedules \( s, t, u, \) and \( s, t, u, \) and all multisets \( M \):

   (1) if \(\langle s_1 ; (s_2 ; s_3) \rangle \overset{\sigma}{\to} \langle t, M' \rangle \). Then this transition is according to either of the following:
4. (S0): Then, $(s_1, M) \xrightarrow{\sigma} (t', M')$ and $t' ; (s_2 ; s_3) \equiv t$. Then according to (S0) we have: $(s_1 ; s_2, M) \xrightarrow{\sigma} (t' ; s_2, M')$ and using (S0) again results in: $((s_1 ; s_2) ; s_3, M) \xrightarrow{\sigma} ((t' ; s_2) ; s_3, M')$. Also, according to the definition of $R$, $((t' ; s_2 ; s_3), M')$, $((t' ; s_1 ; s_2) ; s_3, M') \in R$.

(b) (S1): Then, $(s_1, M) \sqrt{\land}$ and $(s_2 ; s_3, M) \xrightarrow{\sigma} (t, M')$. The $\sigma$ transition of $s_2 ; s_3$ is due to either of the following rules:

i. (S0): Then, $(s_2, M) \xrightarrow{\sigma} (t', M')$ where $t' ; s_2 \equiv t$. According to (S1), $(s_1 ; s_2, M) \xrightarrow{\sigma} (t', M')$ and by using (S0) we have $(s_1 ; s_2) ; s_3, M) \xrightarrow{\sigma} (t' ; s_3, M')$ and since $t' ; s_3 \equiv t$ then the two processes are in the identity subset of $R$.

ii. (S1): Then, $(s_1, M) \sqrt{\land}$, $(s_2, M) \sqrt{\land}$, and $(s_3, M) \xrightarrow{\sigma} (t, M')$. According to (S2), we have $(s_1 ; s_2, M) \sqrt{\land}$. Using rule (S1) results in $((s_1 ; s_2) ; s_3, M) \xrightarrow{\sigma} (t, M')$.

(2) if $((s_1 ; s_2) ; s_3, M) \xrightarrow{\sigma} (t, M')$. Then this transition is according to either of the following:

(a) (S0): Then, $(s_1 ; s_2, M) \xrightarrow{\sigma} (t', M')$ and $t' ; s_2 \equiv t$. The transition of $s_1 ; s_2$ is either due to:

i. (S0): Then, $(s_1, M) \xrightarrow{\sigma} (t'', M')$ where $t'' ; s_2 \equiv t'$. According to (S0), $(s_1 ; s_2, M) \xrightarrow{\sigma} (t'' ; s_2, M')$ and due to the definition of $R$ we have $((t'' ; s_2, s_3) ; s_3, M') \in R$.

ii. (S1): Then $(s_1, M) \sqrt{\land}$ and $(s_2, M) \xrightarrow{\sigma} (t', M')$. According to (S0), we have $(s_2 ; s_3, M) \xrightarrow{\sigma} (t' ; s_2, M')$ and according to (S2), $(s_1 ; s_2, s_3, M) \xrightarrow{\sigma} (t' ; s_3, M')$, and since $t' ; s_3 \equiv t$, this case is also complete.

(b) (S1): Then $(s_1 ; s_2, M) \sqrt{\land}$ and $(s_3, M) \xrightarrow{\sigma} (t, M')$. It follows from (S2) that $(s_1, M) \sqrt{\land}$ and $(s_2, M) \sqrt{\land}$. Hence, according to (S1), we have $(s_2 ; s_3, M) \xrightarrow{\sigma} (t, M')$ and again according to (S1), $(s_1 ; s_2, s_3, M) \xrightarrow{\sigma} (t, M')$.

(3), (4) $((s_1 ; s_2) ; s_3, M) \sqrt{\land} \quad (s_1, M) \sqrt{\land} \quad (s_2, M) \sqrt{\land} \quad (s_3, M) \sqrt{\land} \quad (s_1, M) \sqrt{\land} \quad (s_2, M) \sqrt{\land} \quad (s_3, M) \sqrt{\land} \quad (s_1, M) \sqrt{\land} \quad (s_2, M) \sqrt{\land} \quad (s_3, M) \sqrt{\land}$.

4. We define $R$ as $R \triangleq \{((s, M), (skip || s, M))$, for all schedules $s$ and multisets $M\}$. 

(1) If $(skip || s, M) \xrightarrow{\sigma} (t, M')$, then this transition can be due to

(a) (P0): It cannot be the case because for all $\sigma; (skip, M) \xrightarrow{\sigma}$

(b) (P1): Then, we have $((s, M) \xrightarrow{\sigma} (s', M')$ and $t \equiv skip || s'$, and hence $((s', M'), (t, M')) \in R$. 

76
(c) (P2): It cannot be the case following the same reasoning as in item (a).

(2) If \(\langle s, M \rangle \stackrel{\sigma}{\rightarrow} \langle s', M' \rangle\), then according to (P1) \(\langle \text{skip} \parallel s, M \rangle \stackrel{\sigma}{\rightarrow} \langle \text{skip} \parallel s', M' \rangle\), and we have \(\langle (s', M'), \langle \text{skip} \parallel s', M' \rangle \rangle \in R\).

(3) (4) \(\langle s, M \rangle \sqrt{\checkmark} \Leftrightarrow \langle \text{skip}, M \rangle \sqrt{\checkmark} \Leftrightarrow \langle s, M \rangle \sqrt{\checkmark} \Leftrightarrow \langle \text{skip} \parallel s, M \rangle \sqrt{\checkmark}\)

5. We define the relation \(R\) as \(R \overset{\Delta}{=} \{(\langle s \parallel t, M \rangle, \langle t \parallel s, M \rangle)\}\) for all schedules \(s\) and \(t\) and all multisets \(M\):

(1) \(\langle s_1 \parallel s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t, M' \rangle\); then this transition is due to one of the following rules:

(a) (P0): \(\langle s_1, M \rangle \overset{\rightarrow}{\rightarrow} \langle t', M' \rangle\) and \(t \equiv t' \parallel s_2\); then according to (P1) we have \(\langle s_2 \parallel s_1, M \rangle \overset{\rightarrow}{\rightarrow} \langle s_2 \parallel t', M' \rangle\) and according to the definition of \(R\), \(\langle (t' \parallel s_2, M'), \langle s_2 \parallel t', M' \rangle \rangle \in R\).

(b) (P1): \(\langle s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t, M \rangle\) and \(t \equiv s_1 \parallel t'\); then according to (P0) we have \(\langle s_2 \parallel s_1, M \rangle \overset{\rightarrow}{\rightarrow} \langle t' \parallel s_1, M' \rangle\) and according to the definition of \(R\), \(\langle (t' \parallel s_1, M'), \langle s_2 \parallel t' \parallel s_1 \rangle \rangle \in R\).

(c) (P2): \(\langle s_1, M \rangle \overset{\rightarrow}{\rightarrow} \langle t_1, M_1 \rangle\), \(\langle s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t_2, M_2 \rangle\), with \(M \models \sigma_1 \Rightarrow \sigma_2\), \(\sigma = \sigma_1 \cup \sigma_2\), \(t \equiv t' \parallel t_2\) and \(M' = M[\sigma_1, \sigma_2]\). According to (P2) and proposition 2.3 with changing the order of hypothesis, we have \(\langle s_2 \parallel s_1 \parallel M \parallel \sigma_2, \sigma_1 \rangle \overset{\rightarrow}{\rightarrow} \langle t_2 \parallel s_1 \parallel M \parallel \sigma_2, \sigma_1 \rangle\). But, it follows from proposition 2.5 that \(M[\sigma_1, \sigma_2] = M'\) and hence, \(\langle (s_1 \parallel s_2, M'), \langle s_2 \parallel s_1, M' \parallel s_1 \parallel M' \rangle \rangle \in R\).

(2) Due to symmetry, the same reasoning holds for: \(\langle s_2 \parallel s_1, M \rangle \overset{\rightarrow}{\rightarrow} \langle t, M' \rangle\)

(3),(4) \(\langle s_1 \parallel s_2, M \rangle \sqrt{\checkmark} \Leftrightarrow \langle s_2 \parallel s_1, M \rangle \sqrt{\checkmark} \Leftrightarrow \langle s_1 \parallel s_2, M \rangle \sqrt{\checkmark}\)

6. We define the relation \(R\) as \(R \overset{\Delta}{=} \{(\langle s \parallel t \parallel u, M \rangle, \langle s \parallel t \parallel u, M \rangle)\}\) for all schedules \(s\), \(t\), and \(u\) and all multisets \(M\):

(1) \(\langle s_1 \parallel s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t, M' \rangle\); then this transition is due to either of the following rules:

(a) (P0): Then, \(\langle s_1 \parallel s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t', M' \rangle\) and \(t \equiv t' \parallel s_3\). This transition is according to one of the following rules:

\((\text{P0})\): \(\langle s_1, M \rangle \overset{\rightarrow}{\rightarrow} \langle t', M' \rangle\) where \(t' \equiv t' \parallel s_2\). From (P1) it follows that \(\langle s_1 \parallel s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t' \parallel s_2, M \rangle\) and according to the definition of \(R\), \(\langle (t' \parallel s_2, M'), \langle s_2 \parallel t', M' \rangle \rangle \in R\).

\((\text{P1})\): \(\langle s_2, M \rangle \overset{\rightarrow}{\rightarrow} \langle t', M' \rangle\), where \(t' \equiv s_1 \parallel t'\). From the application of (P0) and (P1) it follows that \(\langle s_2 \parallel s_3, M \rangle \overset{\rightarrow}{\rightarrow} \langle t' \parallel s_2, M \rangle\) and then, \(\langle s_1 \parallel s_2 \parallel s_3, M \rangle \overset{\rightarrow}{\rightarrow} \langle t' \parallel s_3, M \rangle\) and according to the definition of \(R\), we have \(\langle (s_1 \parallel (t' \parallel s_3), M'), \langle s_1 \parallel t', M' \parallel s_3 \parallel M' \rangle \rangle \in R\).
(P2): \((s_1, M) \xrightarrow{\sigma_3} (t_1, M_1)\), and \((s_2, M) \xrightarrow{\sigma_2} (t_2, M_2)\), where \(M = \sigma_1 \bowtie \sigma_2\), and \(\sigma = \sigma_1 \bowtie \sigma_2\), and \(t' \equiv t_1 \parallel t_2\). Applying (P1) and then (P2) results in \((s_1 \parallel s_2, M) \xrightarrow{\tau_3} (t_1 \parallel t_2, M')\), and \((t'_1 \parallel t'_2 \parallel s_3, M')\); \((t_1 \parallel t_2 \parallel s_3, M') \in R\).

(b) (P1) The reasoning follows the same line as in the previous item.

(c) (P2) \((s_1 \parallel s_2, M) \xrightarrow{\tau_3} (t_1, M_1)\), and \((s_2, M) \xrightarrow{\tau_2} (t_2, M_2)\), where \(M = \sigma_1 \bowtie \sigma_2\), \(\sigma = \sigma_1 \bowtie \sigma_2\), \(M = M[\sigma_1, \sigma_2]\), and \(t' \equiv t_1 \parallel t_2\). As in the previous items, the transition of \(\sigma_1\) can be done according to (P1), (P2), or (P2). We just investigate the ease of (P2) since the rest are similar to the previous items.

According to (P2), \((s_1 \parallel s_2, M) \xrightarrow{\tau_3} (t_3, M_3)\), where \(M = \sigma_3 \bowtie \sigma_4\), \(M = M[\sigma_3, \sigma_4]\), and \(t' \equiv t_3\). It follows from Proposition 2.7 and Theorem 2.9 that \(\sigma_1 \bowtie \sigma_2\) and \(M = M[\sigma_3, \sigma_4]\), and thus by applying (P2) twice, we have \((s_1 \parallel s_2 \parallel s_3, M) \xrightarrow{\tau_3 \parallel \tau_2} \langle t_3 \parallel (t_2 \parallel t_3), M[\sigma_3, (\sigma_4, \sigma_2)]\rangle\), but \(\sigma = \sigma_3, (\sigma_4, \sigma_2)\) and \((t_1 \parallel t_2 \parallel t_3, M'), ((t_1 \parallel t_2 \parallel t_3, M') \in R\).

(2) The reasoning of this part is similar to the previous item.

\(7\). Consider the bisimulation relation relating states \(\langle r \bowtie \text{skip}[\text{skip}], M \rangle\) and \(\langle \text{skip}, M \rangle\) for all rules \(r\) and multisets \(M\).

(a) We claim that for no \(\sigma \) and \(M\), \(\langle r \bowtie \text{skip}[\text{skip}], M \rangle\) can perform a \(\sigma\) transition. Suppose that it does and \(\langle r \bowtie \text{skip}[\text{skip}], M \rangle \xrightarrow{\tau_3} (t, M')\), then this transition is due to (RC0) or (RC1). However, since for all \(\sigma\) and \(M\), \(\langle \text{skip}, M \rangle \xrightarrow{\tau_3}\), neither of the cases are possible (thus, the implication in Definition 2.15 holds trivially).

(b) Similarly, for all \(\sigma\) and \(M\), \(\langle \text{skip}, M \rangle \xrightarrow{\tau_3}\).

(c) \(\langle \text{skip}, M \rangle \xrightarrow{\tau_3}\).

\(8\). We define \(R \supseteq \text{id} \cup \{(r \bowtie r[\text{skip}], M), (r, M)\}\), for all \(r\) and \(M\).

(a) (b) \((r \bowtie r[\text{skip}], M) \xrightarrow{\tau_3} (t, M')\), then following a reasoning similar to that of previous item, this transition can only be due to (RC0). Thus, \(\langle r, M \rangle \xrightarrow{\tau_3} (t, M')\).

(b) \((r, M) \xrightarrow{\tau_3} (t, M')\), then according to (RC0) \((r \bowtie r[\text{skip}], M) \xrightarrow{\tau_3} (t, M')\).

(c) \(\langle r \bowtie r[\text{skip}], M \rangle \xrightarrow{\tau_3}\).

\(9\). Consider the minimal bisimulation relation containing identity and pairs of \(\{(r \bowtie s_1, s_2; M), (r \bowtie (s_1 \parallel s_2; s_1 \parallel s_2), M)\}\), for all rules \(r\), schedules \(s_1\), \(s_2\), and \(s_3\), and all multisets \(M\):
(a) \( \{ r \rhd_{s_1[s_2]} ; s_2, M \} \xrightarrow{\ast} (t, M') \); then this transition is due to either of the following rules:

i. (S0): \( \{ r \rhd_{s_1[s_2]} ; s_2, M \} \xrightarrow{\ast} (t', M') \) and \( t \equiv t' ; s_3 \). This transition is due to either (RC0) or (RC1):

- (RC0): \( \neg((r, M) \wedge (s_1, M)) \) and \( (s_1, M) \xrightarrow{\ast} (t', M') \). It follows from (S0) that \( (s_1 ; s_2, M) \xrightarrow{\ast} (t' ; s_3, M') \) and thus according to (RC0), \( (r \rhd_{s_1[s_2]} ; s_2 ; s_3, M) \xrightarrow{\ast} (t, M') \).

- (RC1): \( (r, M) \vee (s_2, M) \) and \( (s_2, M) \xrightarrow{\ast} (t', M') \). It follows from (S0) that \( (s_2 ; s_3, M) \xrightarrow{\ast} (t' ; s_3, M') \) and thus according to (RC1), \( (r \rhd_{s_1[s_2]} ; s_2 ; s_3, M) \xrightarrow{\ast} (t, M') \).

ii. (S1): \( (r \rhd_{s_1[s_2]} ; s_2, M) \) and \( (s_3, M) \xrightarrow{\ast} (t, M') \). The termination of the rule conditional can be due to either of the following rules:

- (RC2): \( \neg((r, M) \wedge (s_1, M)) \) and \( (s_1, M) \xrightarrow{\ast} (t, M') \). It follows from (S1) that \( (s_1 ; s_2, M) \xrightarrow{\ast} (t, M') \) and according to (RC0) \( (r \rhd_{s_1[s_2]} ; s_2 ; s_3, M) \xrightarrow{\ast} (t, M') \).

- (RC3): \( (r, M) \wedge (s_2, M) \) and \( (s_2, M) \xrightarrow{\ast} (t, M') \). It follows from (S1) that \( (s_2 ; s_3, M) \xrightarrow{\ast} (t, M') \) and according to (RC1) \( (r \rhd_{s_1[s_2]} ; s_2 ; s_3, M) \xrightarrow{\ast} (t, M') \).

(b) \( (r \rhd_{s_1[s_2]} ; s_2, M) \xrightarrow{\ast} (t, M') \). Then, this transition is due to one of the following rules:

i. (RC0): \( \neg((r, M) \wedge (s_1, M)) \) and \( (s_1 ; s_3, M) \xrightarrow{\ast} (t, M') \). Then this transition is due to either of the following rules:

- (S0): \( (s_1, M) \xrightarrow{\ast} (t', M') \) and \( t \equiv t' \); \( s_3 \). Thus according to (RC0), \( (r \rhd_{s_1[s_2]} ; s_2, M) \xrightarrow{\ast} (t', M') \) and according to (S0), \( (r \rhd_{s_1[s_2]} ; s_2, M) \xrightarrow{\ast} (t, M') \).

- (S1): \( (s_1, M) \) and \( (s_3, M) \xrightarrow{\ast} (t', M') \). It follows from (RC0) that \( (r \rhd_{s_1[s_2]} ; s_2, M) \) and using (S1) results in \( (r \rhd_{s_1[s_2]} ; s_2, M) \xrightarrow{\ast} (t, M') \).

ii. (RC1): Similar to the previous item.

(c) \( (r \rhd_{s_1[\xi]} ; s_2, M) \) \( \xrightarrow{\ast} (r \rhd_{s_1[s_2]} ; s_2, M) \) \( \wedge (s_2, M) \) \( \wedge (s_3, M) \). This trivially follows from (R0) and (R1).

10. Trivially follows from (R0) and (R1).

### Lemma A.2

For an arbitrary rule \( r_i, r_i \parallel \mu X. r_i \rhd (r_i \parallel X) \equiv \mu X. r_i \rhd (r_i \parallel X) \).
Proof. We define the bisimulation relation $R$, as the smallest set containing $\text{id}$ together with pairs of $(r_{i} \parallel M, \{r_{i} \parallel X\}, M)$, $(\mu X.r_{i} \leadsto (r_{i} \parallel X))$:

1. $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')$. This transition can only be due to \((R0)\), and hence: $(r_{i} \leadsto (r_{i} \parallel X))$, $M \rightarrow (t, M')$. But this transition can only be due to \((RC0)\) (since the second argument of the rule conditional is $\text{skip}$); according to \((RC0)\) this means that $(r_{i} \parallel M)$.

2. $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')$ Then, this transition is according to \((PO), (P1), (P2)\). In the first and the last case, we have that $(r_{i}, M)$ has to make a transition, and hence $\neg(r_{i}, M) \lor$. If the transition is due to \((P1)\), then $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M'' = M'' \lor)$ and again since this transition can only be due to \((R0)\) and \((RC0)\) this means that $(r_{i}, M)$.

Using the original transition of $(r_{i} \parallel M, \{r_{i} \parallel X\}, M) \rightarrow (t, M')$ and according to \((RC0)\), we have $(r_{i} \leadsto (r_{i} \parallel M, \{r_{i} \parallel X\}), M) \rightarrow (t, M')$ and hence, according to \((R0)\) $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')$.

3. $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')$.

4. $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')$.

Corollary A.3 For an arbitrary program $R = \{r_{0}, ..., r_{n}\}$, and a rule $r_{i} \in R$, \((\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')\) fall in this relation. We do this using an induction on the depth of $\sigma$ transition:

1. Induction base: If $(\mu X.r_{i} \leadsto (r_{i} \parallel X), M) \rightarrow (t, M')$, with a proof of depth $3$ (The minimum depth of the proof for this term is $3$, because the recursion and rule conditional should first be resolved), then this transition
can be only due to application of (R0), (RC0), and (P0) subsequently. Hence, \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (\text{skip} \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M') \); then \( t \equiv \text{skip} \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X) \). It follows from Proposition 2.17 that \( (\langle t, M' \rangle, \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle) \in R \). Since \( R \) is a bisimulation relation \( \langle \langle t, M' \rangle, (\mu X.x_i \rightsquigarrow (r_i \parallel X), M) \rangle \in R \), for all \( M \).

2. Induction step: Consider the transition \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (t, M') \) with a proof of depth \( n \), then this transition should be according to (R0) and (RC0) and hence, \( \langle r_i \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (t, M') \). Then, this transition can be according to either of the following rules:

(a) (P0): \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (\text{skip} \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M') \) and \( (\text{skip} \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M') \in R \).

(b) (P1): \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (t' ,M') \) and \( t \equiv t' \). Thus, according to the induction hypothesis, \( t' \equiv \mu X.x_i \rightsquigarrow (r_i \parallel X) \) and since \( R \) is closed under congruence \( \langle r_i \parallel t', M' \rangle, \langle r_i \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle \rangle \in R \). But, according to Lemma A.2, \( \langle r_i \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle \rangle \in R \), hence since \( R \) is closed under transitivity \( \langle t', M' \rangle, \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle \rangle \in R \).

(c) (P2): \( \langle r_i, M \rangle \Rightarrow (\text{skip}, M), \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (t_2, M_2) \), \( \sigma = \sigma_1, \sigma_2 \) and according to the induction hypothesis \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle \rangle \in R \). Thus, \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (\text{skip} \parallel t_2, M_2) \) and \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M', (\text{skip} \parallel t_2, M_2) \rangle \rangle \in R \).

**Corollary A.5** For an arbitrary program \( R = \{ r_0, ..., r_n \} \), if \( \langle \langle \mu X.x_0 \rightsquigarrow (r_0 \parallel X) \rangle \rangle \parallel ... \parallel \langle \mu X.x_n \rightsquigarrow (r_n \parallel X) \rangle \rangle \Rightarrow (t, M') \), then \( t \equiv \langle \mu X.x_0 \rightsquigarrow (r_0 \parallel X) \rangle \parallel ... \parallel \langle \mu X.x_n \rightsquigarrow (r_n \parallel X) \rangle \rangle \).

**Lemma A.6** For an arbitrary rule \( r_i \), \( \mu X.x_i \rightsquigarrow (r_i \parallel X) \equiv \mu X.x_i \rightsquigarrow (r_i \parallel X) \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X) \parallel \).

**Proof.** Consider the bisimulation relation of Lemma A.4 together with pairs of \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X) \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \) and \( (\mu X.x_i \rightsquigarrow (r_i \parallel X), M) \), closed under congruence, symmetry and transitivity (denoted by \( R \)):

1. \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M \rangle \Rightarrow (t, M') \Rightarrow \langle \mu X.x_i \rightsquigarrow (r_i \parallel X) \parallel \mu X.x_i \rightsquigarrow (r_i \parallel X) \parallel t, M' \rangle \). According to Lemma A.4, \( \langle \langle t, M' \rangle, (\mu X.x_i \rightsquigarrow (r_i \parallel X), M') \rangle \rangle \in R \). Also according to the definition of \( R \), \( \langle \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle, (\mu X.x_i \rightsquigarrow (r_i \parallel X), M') \rangle \rangle \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle \rangle \in R \). Since \( R \) is closed under congruence \( \langle \mu X.x_i \rightsquigarrow (r_i \parallel X), M' \rangle \rangle \in R \). \( R \) is closed under transitivity and hence \( (t, M'), \langle \mu X.x_i \rightsquigarrow (r_i \parallel X) \parallel t, M' \rangle \rangle \in R \).
2. \( \mu X.r_1 \prec (r_1 \parallel X) \parallel \mu X.r_1 \prec (r_1 \parallel X), M \xrightarrow{\sigma} (t, M') \) we prove that \( \langle \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle t', M' \rangle \) and \( \langle t, M' \rangle, \langle t', M' \rangle \in R \) by an induction on the depth of the proof for \( \sigma \) transition.

Induction base: The minimum depth of a \( \sigma \) transition is when either of the parallel components \( (\mu X.r_1 \prec (r_1 \parallel X)) \) or both of them perform a single rule transition:

(a) \((P0)\): Then, this transition is due to \((R0)\), \((RC0)\), and \((P0)\); then \( (r_1, M) \xrightarrow{\sigma} (skip, M'), t \equiv \langle \mu X.r_1 \prec (r_1 \parallel X) \parallel \mu X.r_1 \prec (r_1 \parallel X) \rangle \) and so, according to \((P0)\) \( \langle r_1 \parallel \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle skip \parallel \mu X.r_1 \prec (r_1 \parallel X) \rangle \). Thus, it follows from \((RC0)\) and \((R0)\) that \( \langle \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle skip \parallel \mu X.r_1 \prec (r_1 \parallel X), M' \rangle \). According to the definition of \( R \) \( (\mu X.r_1 \prec (r_1 \parallel X), \mu X.r_1 \prec (r_1 \parallel X)) \in R \). Since \( R \) is closed under congruence (\( \langle \mu X.r_1 \prec (r_1 \parallel X), skip \parallel (\mu X.r_1 \prec (r_1 \parallel X) || \mu X.r_1 \prec (r_1 \parallel X) \rangle \in R \).

(b) \((P1)\): Similar to the previous item.

(c) \((P2)\): \( (r_1, M) \xrightarrow{\sigma} (skip, M_1), (r_1, M) \xrightarrow{\sigma} (skip, M_2), M \equiv \sigma_1, \sigma_2 \) and \( \sigma = \sigma_1, \sigma_2 \). Then, according to \((P0)\) and \((RC0)\) \( (r_1 \prec (r_1 \parallel X), M) \xrightarrow{\sigma} \langle skip \parallel \mu X.r_1 \prec (r_1 \parallel X) \rangle, M_1 \) and according to \((R0)\) \( \langle (\mu X.r_1 \prec (r_1 \parallel X), (skip || \mu X.r_1 \prec (r_1 \parallel X), M_1 \rangle \xrightarrow{\sigma} \langle skip \parallel \mu X.r_1 \prec (r_1 \parallel X) \rangle \). Due to \((P2)\) and \((RC0)\) and \( (r_1 \prec (r_1 \parallel X) || \mu X.r_1 \prec (r_1 \parallel X) \rangle, M_1 \rangle \xrightarrow{\sigma} \langle skip \parallel \mu X.r_1 \prec (r_1 \parallel X) \rangle, M_1 \rangle \). Finally, according to \((R0)\), \( \langle (\mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle skip \parallel (\mu X.r_1 \prec (r_1 \parallel X), M \rangle \rangle \} \). Since \( R \) contains pairs of Proposition 2.17 and is closed under transitivity and congruence (\( \langle \mu X.r_1 \prec (r_1 \parallel X), (skip \parallel \mu X.r_1 \prec (r_1 \parallel X) \rangle \parallel \mu X.r_1 \prec (r_1 \parallel X), M' \rangle \rangle \} \in R \).

Induction Step: The transition \( \sigma \) can be due to \((P0)\), \((P1)\), or \((P2)\). Because of symmetry, we only consider the cases of \((P0)\) and \((P2)\):

(a) \((P0)\): Then \( \langle \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle t', M' \rangle \) and \( t \equiv t' \parallel \mu X.r_1 \prec (r_1 \parallel X) \). This completes the case if we show that \( \langle (t', M' \rangle, \langle t, M' \rangle \rangle \in R \).

According to Lemma A.4, \( \langle (t', M' \rangle, \langle t, M' \rangle \rangle \in R \). According to the definition of \( R \), \( (\mu X.r_1 \prec (r_1 \parallel X) \parallel \mu X.r_1 \prec (r_1 \parallel X), M' \rangle, \mu X.r_1 \prec (r_1 \parallel X), M' \rangle) \in R \). Since \( R \) is closed under transitivity and congruence, \( \langle (t' \parallel \mu X.r_1 \prec (r_1 \parallel X), M', \langle t' \parallel \mu X.r_1 \prec (r_1 \parallel X), M' \rangle \rangle \in R \).

(b) \((P1)\): Similar to the previous item.

(c) \((P2)\): \( \langle \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle t_1, M_1 \rangle \) and \( \langle \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle t_2, M_2 \rangle \). Then each of these transitions should be due to \((R0)\) and \((RC0)\) \( \langle r_1 \parallel \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle t_1, M_1 \rangle \) and \( \langle r_1 \parallel \mu X.r_1 \prec (r_1 \parallel X), M \rangle \xrightarrow{\sigma} \langle t_2, M_2 \rangle \).
(r₁ || X), M) ⊮₄ (t₂, M₂). Each of these transition can be due to (P₀), (P₁), or (P₂) leading to 9 different cases. Due to symmetry 6 cases can be differentiated as follows:

i. (P₀) and (P₀) (the same case as item (c) in the induction base):
(r₁, M) ⊮₄ ⟨skip, M₁⟩, (r₁, M) ⊮₄ ⟨skip, M₂⟩, t₁ = t₂ = ⟨skip || μX.r₁ ∩ (r₁ || X)⟩ || ⟨skip || μX.r₁ ∩ (r₁ || X)⟩. Then, according to (P₀) and (RC₀) (r₁ ∩ (r₁ || X), M), M) ⊮₄ ⟨skip || μX.r₁ ∩ (r₁ || X), M₁⟩ and according (R₀) (μX.r₁ ∩ (r₁ || X), M), M) ⊮₄ ⟨skip || μX.r₁ ∩ (r₁ || X), M₂⟩. Due to (P₂) and (RC₀) (r₁ ∩ (r₁ || X), M), M) ⊮₄ ⟨skip || μX.r₁ ∩ (r₁ || X), M₁⟩. Finally, according to (R₀), (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨skip || μX.r₁ ∩ (r₁ || X), M₁⟩ and (⟨skip || μX.r₁ ∩ (r₁ || X), M₁⟩ || ⟨skip || μX.r₁ ∩ (r₁ || X), M₂⟩) ∈ R. It follows from Lemma A.4 that (⟨skip || M₁'), (μX.r₁ ∩ (r₁ || X), M₀) ∈ R. This it follows from Proposition 2.17 and congruence that (⟨skip || M₀'), (t₀)) ∈ R.

ii. (P₀) and (P₁): (r₁, M) ⊮₄ ⟨skip, M₁⟩, (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀, M₂⟩, t₀ = ⟨skip || μX.r₁ ∩ (r₁ || X)⟩ and t₀ = r₁ || t₀₂. Due to (P₂) and (RC₀) (r₁ ∩ (r₁ || X), M), M) ⊮₄ ⟨skip || t₀₂, M₀'⟩ and hence, (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨skip || t₀₂, M₀'). According to Lemma A.2, (⟨t₀₂, M₀'), (t₀₂, M₀')) ∈ R. It follows from Lemma A.4 that (⟨skip || M₀'), (μX.r₁ ∩ (r₁ || X), M₀') ∈ R. This it follows from Proposition 2.17 and congruence that (⟨skip || M₀'), (t₀, M₀')) ∈ R.

iii. (P₀) and (P₂): Similar to item 1 and 2.

iv. (P₁) and (P₁): (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁, M₁⟩, (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₂, M₂⟩, t₀₁ = r₁ || t₀₁ and t₀₂ = r₁ || t₀₂. Due to application of (P₂) that (μX.r₁ ∩ (r₁ || X) || μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁, M₀'⟩ since the transition has a proof of depth n - 1, the induction hypothesis applies. This results in (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁, M₀') where ⟨t₀₂, M₀'), (t₀₂, M₀') ∈ R but according to Lemmas A.2 and A.4 (and due to transitivity), (t₀₁, t₀₂, M₀') ∈ R.

v. (P₁) and (P₂): Similar to item 4.

vi. (P₂) and (P₂): (r₁, M) ⊮₄ ⟨skip, M₁⟩, (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁', M₁⟩, (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₂', M₂⟩, and t₀ = ⟨skip || t₀₁'⟩ || ⟨skip || t₀₂'. First, composing transitions σ₁₂ and σ₂₂ results in (μX.r₁ ∩ (r₁ || X) || μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁', M[σ₁₂, σ₂₂]⟩ and since the new transition has a proof depth less than n, the induction hypothesis applies: (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁', M[σ₁₂, σ₂₂]⟩. According to induction hypothesis, for all M₀', (⟨t₀₁', M₀', (t₀₁', t₀₂', M₀') ∈ R. It follows from Proposition 2.17 that (⟨t₀₁', M₀'⟩, (t₀₁', t₀₂', M₀')) ∈ R. By composing σ₁ and σ₂ and using (RC₀) and (R₀) to the above transition (similar to item 4), we have (μX.r₁ ∩ (r₁ || X), M) ⊮₄ ⟨t₀₁', M₀'⟩.
Lemma A.7 For an arbitrary rule \( r_i \), \( R = \{ r_i \} \equiv \mu X.r_i \models (r_i \parallel X) \).

Proof. We define the bisimulation relation \( Q \) relating pairs of \( (R, M) \) and \( (s, M) \) for all multisets \( M \) and all schedules \( s \). We prove bisimilarity to \( \mu X.r_i \models (r_i \parallel X) \) in the previous lemmas.

1. Induction base: For a \( \sigma \) transition of depth 1, if \( (R, M) \rightarrow_t (R, M') \), then this transition can be only due to (ProgPar), and hence, \( (r_i, M) \rightarrow_1 (t', M') \). Hence, according to (RO), (RC0) and (P0), \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow_1 (t', M') \). According to the Lemma A.4, \( t \) is bisimilar to \( \mu X.r_i \models (r_i \parallel X) \) and thus \( (s, M) \rightarrow_1 (t', M') \).\( (s, M) \rightarrow_1 (t', M') \)?

2. Induction step: Consider the transition \( (R, M) \rightarrow_t (R, M') \) with the proof of depth \( n \). This transition can be only due to (ProgPar) \( (n > 1) \). Hence, \( (R, M) \rightarrow_1 (R, M_1), (R, M) \rightarrow_1 (R, M_2) \), and \( M \models \sigma_1 \rightarrow \sigma_2 \). But according to the induction hypothesis \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow_1 (t'_1, M_1') \) and \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow_1 (t'_2, M_2') \). Thus, according to (P2) and Lemma A.6 \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow_1 (t'', M'') \) and \( t'' \) is bisimilar to \( t'_1 \parallel t'_2 \). According to Lemma A.4, \( t'' \) and thus \( t'_1 \parallel t'_2 \) is bisimilar to \( \mu X.r_i \models (r_i \parallel X) \) and thus \( ((R, M'), (t', M')) \in Q \).

(2) \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow (t', M') \) then \( (R, M) \rightarrow (R, M') \). This statement can be proved following a similar reasoning to that of the previous item (by an induction on the \( \sigma \) transition proof depth).

(3) \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow_1 (t', M') \) then \( (R, M) \rightarrow_1 (R, M') \).

(4) \( (\mu X.r_i \models (r_i \parallel X), M) \rightarrow_1 (t', M') \) then \( (R, M) \rightarrow_1 (R, M') \).
Lemma A.8 For all programs \( R = \{ r_0, ..., r_n \} \) and multisets \( M, (MGS(R), M) \uparrow \) if and only if \( (r_0, M) \uparrow, ..., \) and \( (r_n, M) \uparrow \).

Proof:

1. \( \Rightarrow \): Suppose that \( (MGS(R), M) \uparrow \), then according to (P3), for all \( r_i, \langle \mu X. r_i \Rightarrow (r_i \parallel X) \rangle, M \rangle \uparrow \). Then according to Lemma A.7, \( (r_i, M) \uparrow \).

2. \( \Leftarrow \): Consider an arbitrary rule \( r_i \in R \), \( (r_i, M) \uparrow \); It follows from (RC3) that \( (r_i \parallel (r_i \mid \mu X. r_i \Rightarrow (r_i \parallel X)), M) \uparrow \). Then, it follows from (R1) that \( (\mu X. r_i \Rightarrow (r_i \parallel X), M) \uparrow \). By applying the same procedure on other rules and using (P3), we have \( (MGS(R), M) \uparrow \). \( \square \)

Theorem 2.18 Most General Schedule: For an arbitrary GAMMA program \( R_{n+1} = \{ r_0, ..., r_n \}, ((\mu X. r_0 \Rightarrow (r_0 \parallel X)) \parallel ... \parallel (\mu X. r_n \Rightarrow (r_n \parallel X)) \Leftrightarrow \equiv R_{n+1} \).

Proof. The bisimulation relation of this relation contains pairs relating \( (R_{n+1}, M) \) and \( (t, M) \) for all schedules \( t \) that are proven bisimilar to \( MGS(R_{n+1}) \) (through Lemmas A.2, A.4 and A.6 ). We prove the theorem by an induction on the size of the program (the number of rules):

1. Induction base: For \( n = 0 \), we have \( \mu X. r_0 \Rightarrow (r_0 \parallel X) = MGS(R_0) \equiv R_0 \) according to Lemma A.7.

2. Induction step: We prove that if for a program of size \( n \), \( MGS(R_n) \equiv R_n \), then for a program of size \( n + 1 \), \( MGS(R_{n+1}) = MGS(R_n) \parallel \mu X. r_n \Rightarrow (r_n \parallel X) \equiv R_{n+1} \):

   1. \( (MGS(R_{n+1}), M) \Leftrightarrow \equiv (t, M') \). Then, this transition is due to either of the following rules:

   i. (P0): \( (MGS(R_n), M) \Leftrightarrow \equiv (t', M') \) and \( t \equiv t' \parallel \mu X. r_n \Rightarrow (r_n \parallel X) \). Then, according to the induction step \( (R_n, M) \Leftrightarrow \equiv (R_n, M') \) and according to Lemma 2.10 \( (R_{n+1}, M) \Leftrightarrow \equiv (R_{n+1}, M') \). On the other hand, it follows from Corollary A.5 that \( t' \equiv (\mu X. r_0 \Rightarrow (r_0 \parallel X)) \parallel ... \parallel (\mu X. r_n \Rightarrow (r_n \parallel X)) \) and hence, \( ((R_{n+1}, M'), (t, M')) \in Q \).

   ii. (P1): \( (\mu X. r_n \Rightarrow (r_n \parallel X), M) \Leftrightarrow \equiv (t', M') \) and \( t \equiv (\mu X. r_0 \Rightarrow (r_0 \parallel X)) \parallel ... \parallel (\mu X. r_n \Rightarrow (r_n \parallel X))\). Then, according to Lemma A.7 \( (\{ r_n \}, M) \Leftrightarrow \equiv (\{ r_n \}, M') \). By Lemma 2.12, we will have \( (R_{n+1}, M) \Leftrightarrow \equiv (R_{n+1}, M') \). According to Lemma A.4, \( t' \equiv \mu X. r_n \Rightarrow (r_n \parallel X) \) and hence, \( ((R_{n+1}, M'), (t, M')) \in Q \).

   iii. (P2): \( (MGS(R_n), M) \Leftrightarrow \equiv (t_1, M_1) \) and \( (\mu X. r_n \Rightarrow (r_n \parallel X), M) \Leftrightarrow \equiv (t_2, M_2) \). Then, according to the induction step \( (R_n, M) \Leftrightarrow \equiv (R_n, M_1) \) and according to Lemma A.7, \( (\{ r_n \}, M) \Leftrightarrow \equiv (\{ r_n \}, M_2) \).

85
Applying Lemma 2.12, will result in \( (R_{n+1}, M) \xrightarrow{\ast} (R_{n+1}, M') \).

Again \( (R_{n+1}, M'), (t, M') \in Q \) according to Lemma A.4 and Corollary A.5.

2. Suppose that \( (R_{n+1}, M) \xrightarrow{\ast} (R_{n+1}, M') \). Then according to Lemma 2.12 for the rule \( r_n \), one of the following cases hold:
   i. \( \langle \{r_n\}, M \rangle \xrightarrow{\ast} \langle \{r_n\}, M' \rangle \), then \( \mu X.r_n \rightarrow (r_n \parallel X), M \xrightarrow{\ast} \langle t, M' \rangle \), and according to (P1) \( ((\mu X.r_0 \rightarrow (r_0 \parallel X)) \parallel \ldots \parallel (\mu X.r_n \rightarrow (r_n \parallel X))) \xrightarrow{\ast} (t, M) \), and according to Corollary A.5, \( ((\mu X.r_0 \rightarrow (r_0 \parallel X)) \parallel \ldots \parallel (\mu X.r_n \rightarrow (r_n \parallel X))) \) is bisimilar to \( ((\mu X.r_0 \rightarrow (r_0 \parallel X)) \parallel \ldots \parallel t) \). Hence, \( ((R_{n+1}, M'), \langle (\mu X.r_0 \rightarrow (r_0 \parallel X)) \parallel \ldots \parallel t, M' \rangle) \in Q \).
   ii. \( \langle \{r_n\}, M \rangle \xrightarrow{\ast} \langle \{r_n\}, M_1 \rangle, \langle R_n, M_2 \rangle \xrightarrow{\ast} \langle R_n, M_2 \rangle, M = \sigma_1 \bowtie \sigma_2 \), and \( \sigma = \sigma_1, \sigma_2 \). Then according to Lemma A.7, \( (\mu X.r_n \rightarrow (r_n \parallel X), M) \xrightarrow{\ast} \langle t_1, M_1 \rangle \) and according to the induction hypothesis, \( MGS(R_n) \xrightarrow{\ast} \langle t_2, M_2 \rangle \). According to (P2), \( MGS(R_n) \parallel \mu X.r_n \rightarrow (r_n \parallel X), M \xrightarrow{\ast} \langle t_1 \parallel t_2, M' \rangle \). It follows from Lemma A.5 and Corollary A.5 that \( ((R_{n+1}, M'), \langle t_1 \parallel t_2, M' \rangle) \in Q \).
   iii. \( \langle R_n, M \rangle \rightarrow (R_n, M') \) and according to the induction hypothesis, \( MGS(R_n), M \xrightarrow{\ast} \langle t', M' \rangle \). It follows from (P0) that \( MGS(R_n) \parallel \mu X.r_n \rightarrow (r_n \parallel X), M \xrightarrow{\ast} \langle t' \parallel (\mu X.r_n \rightarrow (r_n \parallel X), M') \rangle \). According to A.5, we have \( t' \parallel \mu X.r_n \rightarrow (r_n \parallel X) \equiv MGS(R_{n+1}) \) and thus \( ((R_{n+1}, M'), \langle t' \parallel \mu X.r_n \rightarrow (r_n \parallel X), M' \rangle) \in Q \).

3, 4. It follows from Lemma A.8 and (ProgTerm) that \( MGS(R_{n+1}), M \xrightarrow{\ast} \Rightarrow (R_{n+1}, M) \).
Proposition 2.19 For two arbitrary GAMMA programs $R_1$ and $R_2$ and two arbitrary schedules $s$ and $t$, if $R_1 \equiv s$ and $R_2 \equiv t$, then $R_1 \sqcup R_2 \equiv s \parallel t$.

Proof.

1. $\langle s \parallel t, M \rangle \xrightarrow{s'} \langle s', t', M' \rangle$; then, this transition is due to either of the following rules:

(a) (P0): $\langle s, M \rangle \xrightarrow{s'} \langle s', M' \rangle$ and $t' = t$, then $s \equiv R_1$, $\langle R_1, M \rangle \xrightarrow{s'} \langle R_1, M' \rangle$. It follows from Corollary 2.11 that $\langle R_1 \sqcup R_2, M \rangle \xrightarrow{s'} \langle R_1 \sqcup R_2, M' \rangle$ and since $s \equiv R_1$, it should hold that $s' \equiv R_1$ and since bisimilarity is congruence, $s' \parallel t \equiv R_1 \sqcup R_2$.

(b) (P1): Similar to the previous item.

(c) (P2): $\langle s, M \rangle \xrightarrow{s'} \langle s', M' \rangle$, $\langle t, M \rangle \xrightarrow{t'} \langle t', M_2 \rangle$, and $M \models \sigma_1 \parallel \sigma_2$. Then, since $s \equiv R_1$ and $t \equiv R_2$, $\langle R_1, M \rangle \xrightarrow{s'} \langle R_1, M_1 \rangle$, $\langle R_2, M \rangle \xrightarrow{t'} \langle R_2, M_2 \rangle$. It follows from Corollary 2.11 and (ProgPar) that $\langle R_1 \sqcup R_2, M \rangle \xrightarrow{s'} \langle R_1 \sqcup R_2, M' \rangle$, also since $s' \equiv R_1$ and $t' \equiv R_2$, it follows from $s \parallel t \equiv R_1 \sqcup R_2$ that $s' \parallel t' \equiv R_1 \sqcup R_2$.

2. $\langle R_1 \sqcup R_2, M \rangle \xrightarrow{s'} \langle R_1 \sqcup R_2, M' \rangle$ then considering $R_2$, according to Lemma A.9, one of the following cases holds:

(a) $\langle (R_1 \cap R_2), M \rangle \xrightarrow{s'} \langle (R_1 \cap R_2), M' \rangle$ then according to Corollary 2.11 $\langle R_1, M \rangle \xrightarrow{s'} \langle R_1, M' \rangle$ and since $s \equiv R_1$, $\langle s, M \rangle \xrightarrow{s'} \langle s', M' \rangle$ and
\[ s' \cong R_1. \] It follows from (P1) that \( \langle s || t, M \rangle \overset{\sigma_1}{\rightarrow} \langle s' || t, M' \rangle \), and \( s' || t \cong R_1 \cup R_2 \).

(b) \( \langle (R_1 \cup R_2) \setminus R_2, M \rangle \overset{\sigma_2}{\rightarrow} \langle (R_1 \cup R_2) \setminus R_2, M' \rangle, \langle R_2, M \rangle \overset{\sigma_2}{\rightarrow} \langle R_2, M' \rangle, \sigma = \sigma_1, \sigma_2, \) and \( M \models \sigma_1 \Rightarrow \sigma_2 \).

Then, due to Corollary 2.11 \( \langle R_1, M \rangle \overset{\sigma_1}{\rightarrow} \langle R_1, M' \rangle \) and \( \langle R_2, M \rangle \overset{\sigma_2}{\rightarrow} \langle R_2, M' \rangle \). Since \( s \cong R_1 \) and \( t \cong R_2 \), \( \langle s, M \rangle \overset{\sigma_1}{\rightarrow} \langle s', M' \rangle \) and \( \langle t, M \rangle \overset{\sigma_2}{\rightarrow} \langle t', M' \rangle \). It follows from (P2) that \( \langle s || t, M \rangle \overset{\sigma_1}{\rightarrow} \langle s' || t', M' \rangle \) and \( s' || t' \cong R_1 \cup R_2 \).

(c) \( \langle R_2, M \rangle \overset{\sigma_1}{\rightarrow} \langle R_2, M' \rangle \) then similar to item 1, \( \langle s || t, M \rangle \overset{\sigma_1}{\rightarrow} \langle s || t', M' \rangle \) and \( s || t' \cong R_1 \cup R_2 \).

3. 4. \( \langle R_1 \cup R_2, M \rangle \models \forall r \in R_1 \cup R_2; \langle r, M \rangle \models_1 \iff \forall r \in R_1; \langle r, M \rangle \models_1 \land \forall r \in R_2; \langle r, M \rangle \models_1 \iff \langle R_1, M \rangle \models_1 \land \langle R_2, M \rangle \models_1 \iff \langle s, M \rangle \models_1 \land \langle t, M \rangle \models_1 \iff \langle s || t, M \rangle \models_1. \)
Appendix B

Proofs of Timed Properties

In some cases, to prove properties of timed schedules, we have to be sure that annotations in the schedule term are in line with the tasks in the corresponding multiset. So, we assume that a state is valid if it has this property. Nevertheless, by starting from an empty multiset and a syntactically valid schedule (which is obviously not annotated), we will only reach states that are valid in the above sense because the only rules that change the task multiset, change the annotations to keep them valid, either.

Lemma 4.8 For an arbitrary schedule $s$, an arbitrary task set $T$, and a multiset $M$, if $(s, M, T) \xrightarrow{\delta} (s', M', T')$ and $T$ is consistent with respect to $M$, then $T'$ is consistent with respect to $M'$.

Proof. The proof is done by an induction on the depth of the proof for the $\chi$ transition. For the proof of depth one, the transition can be done only due to the semantic rules of basic computation (Figure 4.5), but all rules of this figure have only one or no task in the task multiset (the scheduled task in the multiset is consistent due to the Definition 2.14 (Enahling valuation)).

The induction step proceeds by analyzing the structure of the schedule performing the transition. Suppose that $(s, M, T) \xrightarrow{\delta} (s', M', T')$ by a proof of depth $n + 1$.

If $s$ is a rule-conditional, sequential, or recursive schedule, then the transition should be according to the semantic rules (TRC0), (TRC1), (TS0), (TS1), or (TR0) (see Figures 4.6 and 4.7), and the premises of these rules contain a transition with the same multiset and task multisets. Since the transition in the premises has a proof of (at most) depth $n$, according to induction step the resulting tasks in $T'$ (which remain the same in the deduction) are consistent with respect to $M'$, too.

If $s$ is a parallel schedule, then the transition is due to (TP0)-(TP3). Rules (TP0) and (TP1) observe the consistency condition. Rules (TP2) and (TP3) do not result in the introduction of any new task. Hence, application of these rules cannot cause inconsistency (in case of task execution transitions, the case follows trivially from Lemma 4.3, and in case of time-change transitions, the corresponding composed substitutions do not change at all).
Using the above lemma, in all the of the following proofs we assume the task multisets to be consistent. As another simplification in our proofs of bisimulation, we rule out analyzing idling transitions since for all schedules (and GAMMA programs) the same idling transitions are applicable.

**Theorem Congruence:** Bisimilarity is a congruence with respect to all schedule operators.

**Proof.** To prove the theorem we should prove the following propositions: For all closed schedules \( s, s', t, \) and \( t' \) and an rules \( r \) and \( r' \), if \( r \cong_t r' \), \( t \cong_t t' \) and \( s \cong_t s' \) then:

1. \( r \circ s[t] \cong_t r' \circ s'[t'] \)
2. \( s ; t \cong_t s' ; t' \)
3. \( s \parallel t \cong_t s' \parallel t' \)
4. \( \mu x.s \cong_t \mu x.s' \)

The proofs of the above propositions follow the same line as those of the un-timed one. There are only two changes necessary to make the proofs valid in timed settings:

1. Changing the bisimulation relation and states to contain annotated schedules and task multisets and changing transition labels to range over both time pass (scheduling) and computation.
2. Analyzing the parallel composition transition to cover the new parallel time pass rule (TP2).

For an example of these changes, see the next proposition. \( \square \)

**Proposition B.1** Some timed bisimilar schedules: According to the semantics of the timed-coordination language, the following bisimilarities hold, for all schedules \( s, s_1, s_2, s_3 \) and rule \( r \):

1. \( \text{skip} ; s \cong_t s \)
2. \( s ; \text{skip} \cong_t s \)
3. \( s_1 ; (s_2 ; s_3) \cong_t (s_1 ; s_2) ; s_3 \)
4. \( \text{skip} \parallel s \cong_t s \)
5. \( s_1 \parallel s_2 \cong_t s_2 \parallel s_1 \)
6. \( (s_1 \parallel s_2) \parallel s_3 \cong_t s_1 \parallel (s_2 \parallel s_3) \)
7. \( r \ni \text{skip}[\text{skip}] \cong_1 \text{skip} \)

8. \( r \ni r[\text{skip}] \cong_1 r \)

9. \( (r \ni s_1[s_2]) ; s_2 \cong_1 (r \ni s_1 ; s_2)[s_2 ; s_3] \)

10. If \( y \) is not present in \( s \), \( \mu x.s \cong_1 \mu y.s[y/x] \).

**Proof.** Apart from the syntactic changes in states and relation, the way to prove items 1, 2, 3, 7, 8, 9, and 10 remain the same as untimed proofs of Proposition 2.17 in Appendix A. Proofs of items 4, 5, and 6 change a bit. To sketch the changes, we give the proof for 5:

We define the relation \( R \) as \( R \equiv \{ ((s,lVI,T) , (t \ni s,M,T)) \mid \text{id for all schedules } s \text{ and } t \text{ (possibly annotated with respect to } T), \text{ multisets } M, \text{ and task multisets } T \} \)

1. \((s_1 \ni s_2,M,T) \not\rightarrow (t,M',T') \); then this transition is due to one of the following rules:

1. (TP0): \((s_1,M,T_1) \not\rightarrow (t',M',T'_1) \). If \( T_2 = T \boxplus T_1 \), then \( T = T_1 \boxplus T_2 \), \( T' = T'_1 \boxplus T_2 \), and \( t \equiv t' \mid s_2 \). Then, according to (TP1), \((s_2 \ni s_1,M,T_2 \boxplus T_1) \not\rightarrow (s_2 \ni t',M',T_2 \boxplus T_1) \) and according to the definition of \( R \), \((t' \mid s_2,M',[s_2,T_2 \boxplus T_1])(s_2 \ni t',M',T_2 \boxplus T_1) \) \( \in R \).

2. (TP1): Due to symmetry, similar to the previous item.

3. (TP2): \((s_1,M,T_1) \not\rightarrow (t'_1,M,T'_1) \), \( M' = M \), \((s_2,M,T_2) \not\rightarrow (t'_2,M,T'_2) \), \( T = T_1 \boxplus T_2 \), \( M' = M' \), \((s_2 \ni s_1,M,T) \not\rightarrow (t'_2,M,T'_2) \). According to (TP2), with changing the order of the hypotheses, we have \((s_2 \ni s_1,M,T) \not\rightarrow (t'_2,M',T'_2) \) and \((s_2 \ni s_1,M,T) \not\rightarrow (t'_2,M',T'_2) \) \( \in R \).

4. (TP3): \((s_1,M,T_1) \not\rightarrow (t'_1,M,T'_1) \), \((s_2,M,T_2) \not\rightarrow (t'_2,M,T'_2) \), \( T = T_1 \boxplus T_2 \), \( T' = T'_1 \boxplus T'_2 \), \( t \equiv t'_1 \mid s_2 \) \( \sigma_1,\sigma_2 \), \( \chi = \sigma_1,\sigma_2 \) and \( M' = M'[\sigma_1,\sigma_2] \). According to (TP3), with changing the order of the hypotheses, we have \((s_2 \ni s_1,M,T) \not\rightarrow (t'_2,M',T'_2) \) \( \subseteq \{ t'_2 \mid t'_1 \} \subseteq \{ t'_2 \mid M'[\sigma_2,\sigma_1] , T' \} \). But, it follows from Lemma 2.5 that \( M[\sigma_1,\sigma_2] = M'[\sigma_2,\sigma_1] \) and hence, \((s_2 \ni s_1,M',T') \) \( \in R \).

Due to symmetry, the same reasoning holds for \((s_2 \ni s_1,M,T) \not\rightarrow (t,M',T') \).

Lemma B.2 For all transitions of the form \((\mu X.r \ni (r \ni X)) \) \( \subseteq (r[r_0 \ni t_0 : I] \ni ... \ni r[r_{m-1} \ni t_{m-1} : I],M,T) \not\rightarrow (t,M',T') \), where \( T = [r_0 \ni t_0 : I,...,r_{m-1} \ni t_{m-1} : I] \) and \( T' = [r_0 \ni t_0' : I,...,r_{m-1} \ni t_{m-1}' : I] ; t \equiv_1 (\mu X.r \ni (r \ni X)) \) \( \subseteq (r[r_0 \ni t_0' : I] \ni ... \ni r[r_{m-1} \ni t_{m-1}' : I]) \).
Proof. We define bisimulation $R$ as the union of the identity relation and the bisimulation relations in the previous lemma, closed under congruence. (Note that we do not have to prove that this is a bisimulation relation.) The $\chi$ transition can be due to one of the following cases:

1. (TP0): $\langle \mu X. r \bowtie (r \parallel X), M, T_1 \rangle \xrightarrow{\ell} \langle t', M', T'_1 \rangle$. But the only transition that can be taken is scheduling a new task (the schedule term in the left-hand side does not have any annotated task to allow time pass or commitment). Thus, $\chi = 0$ and $T'_1 = T_1 \parallel [\sigma' @ 0 : I]$. But then, the whole transition is of the form: $(\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \xrightarrow{\ell} \langle ((\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \parallel [\sigma' @ 0 : I] \rangle$ and $\langle ((\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \parallel [\sigma' @ 0 : I] \rangle, (\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \parallel [\sigma' @ 0 : I] \rangle \rangle \in R$. Since $R$ is a bisimulation relation, this completes the case.

2. (TP1): Then the transition can be due to a number of time pass transitions or a number of task commitments. So, in general the transition will be of one of the following forms:

   (a) $\langle (\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \xrightarrow{\ell} \langle (\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \parallel [\sigma' @ 0 : I] \rangle$, (several tasks might have increased their execution time by $t')$, $T' = [r[\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n + t' : I] | ... | \sigma_n @ t_n : I]$. Hence, the lemma holds trivially (due to the identity part of $R$).

(b) $\langle (\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]), M, T) \xrightarrow{\ell} \langle (\mu X. r \bowtie (r \parallel X)) \parallel ([\sigma[0 @ 0 : I] | ... | (\text{skip} | ...)) \parallel [r[\sigma_n @ t_n : I] | ... | \sigma_n @ t_n : I], M[\sigma], T') \rangle$, and $T'$ is the result of removing the committed tasks from $T$. The lemma again holds since $t$ only has skip schedules instead of committed tasks that are absent in $T'$.

3. (TP2), (TP3): Transition according to these rules is not possible since the first schedule $\mu X. r \bowtie (r \parallel X)$ does not contain any active task set to perform a time pass or commitment.

\begin{lemma} \text{Lemma B.3} \end{lemma} For all $T = \parallel [\sigma[0 @ 0 : I] | ... | \sigma_n @ t_n : I]$, $\mu X. r \bowtie (r \parallel X) \parallel ([r[\sigma[0 @ 0 : I] | ... | r[\sigma_n @ t_n : I]) \bowtie (t, M', T')$. The transition can be due to one of the following rules:

1. (TP0): Then the transition can be only scheduling a new task since there is no active task associated with the component on the left-hand
side of the outermost parallel composition. Hence the transition will be of the form: 
\langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \xrightarrow{\delta} \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r \parallel ...) \parallel r[\sigma_0@a@t_0 : I] : I), M, [\sigma'@0 : I, \sigma_0@t_0 : I, ..., \sigma_n@a@t_n : I] \rangle. But then, the same transition can be done by \mu X.r \rightharpoonup (r \parallel X) \parallel \mu X.r \rightharpoonup (r \parallel X) by just unfolding one of the two parallel components.

2. (TP1): Then the transition is due to time pass and commitment of the task set which is the same for both schedules and hence the transition can be mimicked exactly by \langle \mu X.r \rightharpoonup (r \parallel X) \parallel \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle. Then the transition is due to time pass and task commitment.

3. (TP2), (TP3): No transition is possible due to these rules because the first part (\mu X.r \rightharpoonup (r \parallel X)) is not able to perform time pass or task commitment.

(2) Similar to the previous item.

(3), (4) \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \xrightarrow{\delta} R = \emptyset \wedge \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \xrightarrow{\delta} \langle \mu X.r \rightharpoonup (r \parallel X) \parallel \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle. But then, the same transition can be done by \mu X.r \rightharpoonup (r \parallel X) \parallel \mu X.r \rightharpoonup (r \parallel X) by just unfolding one of the two parallel components.

Note that separating the scheduling from time pass and commitment simplifies the proof significantly compared to the un-timed settings (Lemma A.6).

Lemma B.4 For all M and T = [\sigma_0@t_0 : I, ..., \sigma_n@a@t_n : I], there is a bisimulation relation R such that \langle \{r\}, M, T \rangle, \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \in R.

Proof. We assume the minimum bisimulation relation R containing the pairs of \langle \{r\}, M, T \rangle and \langle t, M, T \rangle for all schedules related to \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle using identity and the bisimulation relations of Lemma B.3 and Proposition B.1, closed under congruence.

1. Induction base: Suppose that the \chi transition is of length one. If the transition is commitment of a task, then the transition can be only due to \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \xrightarrow{\delta} \langle \mu X.r \rightharpoonup \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle. Then it follows from (TP1) that \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \xrightarrow{\delta} \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle and due to (TP1), \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle \xrightarrow{\delta} \langle \mu X.r \rightharpoonup (r \parallel X) \parallel (r[\sigma_0@t_0 : I] \parallel ...) \parallel r[\sigma_n@a@t_n : I] : I), M, T \rangle. The rest of the proof (for the cases of time-pass and scheduling transition) for the induction base is similar to above.
2. For the induction step, the transition is due to either of the following rules:

(a) \textbf{(ProgComp0)}: See induction base.

(b) \textbf{(ProgComp1)}: \{(r), M, T_1\} \xrightarrow{\sigma} \{(r), M'_1, T'_1\} and \{(r), M, T \boxplus T_1\} \xrightarrow{\sigma} \{(r), M'_2, T'_2\}. Thus, according to induction step \((\mu X . x \triangleq (r \parallel X)) || ((r[\sigma_0 : t_{10} : I] \parallel ...) || r[\sigma_{1m} : t_{1m} : I], M, T_1) \xrightarrow{\sigma} ((\mu X . x \triangleq (r \parallel X)) || (r[\sigma'_0 : t'_{10} : I] \parallel ...), M'_1, T'_1)\) and \((\mu X . x \triangleq (r \parallel X)) || ((r[\sigma_0 : t_{20} : I] \parallel ...) || r[\sigma_{2m} : t_{2m} : I], M, T_2) \xrightarrow{\sigma} ((\mu X . x \triangleq (r \parallel X)) || (r[\sigma'_0 : t'_{20} : I] \parallel ...), M'_2, T'_2)\). It follows from (TP3) that \((\mu X . x \triangleq (r \parallel X)) || \mu X . x \triangleq (r \parallel X)) || ((r[\sigma_0 : t_0 : I] \parallel ...) || r[\sigma_n : t_n : I])\).

(c) Cases (ProgTerm0) and (ProgSched) are similar to item 1 and that of (ProgTerm1) is similar to item (b).

Then the transition is due to one of the following rules:

1. \textbf{(TP0)} Then \((\mu X . x \triangleq (r \parallel X), M, T_1) \xrightarrow{\sigma} (t, M, T')\). This transition can be only due to scheduling of a new task and hence, \(T' = T \boxplus [\sigma'_0 @ t'_0 : I], (r, M, \emptyset) \xrightarrow{\sigma_0} (r[\sigma'_0 @ t'_0 : I], M, [\sigma'_0 @ t'_0 : I])\). According to Lemma B.2, \(t \xrightarrow{\sigma_0} \mu X . x \triangleq (r \parallel X) \parallel (r[\sigma'_0 @ t'_0 : I] \parallel ...)\), where \(T' = [\sigma'_0 @ t'_0 : I], \ldots\). It follows from (TCoordSched) that \((r, M, \emptyset) \xrightarrow{\sigma_0} (r, M, [\sigma'_0 @ t'_0 : I])\).

Then, according to (ProgComp) \(\{(r), M, T\} \xrightarrow{\sigma} \{(r), M, T'\}\).

2. \textbf{(TP1)} Similar to the proof of Lemma B.2.

For brevity in presentation, we define the following schedule term: \(TSMGS(r, T) \triangleq (\mu X . x \triangleq (r \parallel X) || (r[\sigma_0 @ t_0 : I] \parallel ...) || r[\sigma_n @ t_n : I])\).

Corollary B.5 For an arbitrary rule \(r\), \{r\} \xrightarrow{\sigma} TSMGS(r, \emptyset).
Corollary B.6 For all transitions of the form $\langle TSMGS(r_0, T_0) \parallel ... \parallel TSMGS(r_n, T_n), M, T_0 \parallel ... \parallel T_n \rangle \xrightarrow{t} \{t, M', T_0' \parallel ... \parallel T_n'\}$, then $t \equiv_{t} TSMGS(r_0, T_0') \parallel ... \parallel TSMGS(r_n, T_n')$.

Lemma B.7 For a timed-GAMMA program $R$, $\langle R, M, T \rangle \xrightarrow{t} \langle R', M', T' \rangle$ if and only if for all rules $r \in R$, one of the following cases holds:

1. $\langle R \setminus \{r\}, M, T \rangle \xrightarrow{t} \langle R \setminus \{r\}, M', T' \rangle$
2. $\langle R \setminus \{r\}, M, T \rangle \xrightarrow{t} \langle \{r\}, M, T' \rangle$ and $M' = T_1' \parallel T_2'$.
3. $\langle \{r\}, M, T \rangle \xrightarrow{t} \langle \{r\}, M', T' \rangle$

Proof. Similar to untimed case proof (Lemma 2.12), by an induction on the depth of the proof.

Theorem 4.9 For an arbitrary timed-GAMMA program $R$, $R \equiv_{t} ((\mu X. r_0 \triangleleft (r_0 \parallel X)) \parallel ... \parallel (\mu X. X))$.

Proof. We construct a bisimulation relation $R$ containing pairs of the form $\langle (R, M, T), (\|_{i \leq n}^r (\mu X. r_i \triangleleft (r_i \parallel X)) \parallel (r_i[\sigma_{in} \triangleleft t_{in} : I_i] \parallel ... \parallel r_i[\sigma_{in} \triangleleft t_{in} : I_i]), M, T_{r_0} \parallel ... \parallel T_{r_n} \rangle \xrightarrow{t} \langle t, M, T' \rangle$. It follows from decomposing the transition (according to (P0), (P1), (P2) or (P3)) and applying the induction hypothesis.

1. Induction base: For a program $\{r\}$ with single rule $r$, it follows from Lemma B.4.
2. Induction step:

1. If $\langle R, M, T \rangle \xrightarrow{t} \langle R', M', T' \rangle$ then according to Lemma B.7, and using either (P0), (P2) or (P3) will prove the hypothesis.
2. $\langle \|_{i \in R} (\mu X. X \triangleleft (r_i \parallel X)) \parallel (r_i[\sigma_{in} \triangleleft t_{in} : I_i] \parallel ... \parallel r_i[\sigma_{in} \triangleleft t_{in} : I_i]), M, T_{r_0} \parallel ... \parallel T_{r_n} \rangle \xrightarrow{t} \langle t, M, T' \rangle$. It follows from decomposing the transition (according to (P0), (P1), (P2) or (P3)) and applying the induction hypothesis.
3. $\langle R, M, T \rangle \not\xrightarrow{t} \langle t, M, T' \rangle$ and for all $r \in R$, $\langle \{r\}, M, T \rangle \not\xrightarrow{t} \langle \{r\}, M, T \rangle$. It follows from decomposing the transition (according to (P0), (P1), (P2) or (P3)).

\[\Box\]