MASTER

Conservative and compositional modeling in the CompSOC platform

Milea, T.Y.

Award date:
2012

Link to publication
Conservative and Compositional Modeling in the CompSOC Platform

Tal Milea

submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE in
COMPUTER SCIENCE AND ENGINEERING

Graduation tutors:
Prof. Kees Goossens
Dr. Marc Geilen
Electronic Systems Group
Department of Electrical Engineering

Graduation supervisor:
Dr. Mohammad Mousavi
Model Driven Software Engineering Group
Department of Mathematics and Computer Science

Eindhoven, October 2012
“The rule is, jam tomorrow and jam yesterday – but never jam today.”
“It must come sometimes to ‘jam today’,” Alice objected.
“No, it can’t,” said the Queen. “It’s jam every other day: today isn’t any other day, you know.”

LEWIS CARROLL, THROUGH THE LOOKING GLASS
Acknowledgments

I would like to thank Prof. Kees Goossens and Dr. Marc Geilen for hosting me in the Electronic Systems Group, where this thesis took place. I am grateful to Dr. Geilen for his endless patience and the always valuable, constructive, and precise feedback. I thank Prof. Goossens for his many insights and suggestions.

I want to especially thank Dr. Mohammad Mousavi for his supervision, for his assistance, and for his involvement in my thesis.

I am thankful to all my family and friends, old and new, for their moral support throughout this research. Finally, I want to thank Uri for his endless love and support, and for his superpower known as the ability to simply make me laugh. Thank you.

Tal Milea
October 2012
Abstract

In the CompSOC platform applications are conservatively modeled and analyzed to achieve strict real-time guarantees. Time-non-deterministic system-components are abstracted to their worst-case temporal behavior, resulting in a deterministic Synchronous Dataflow (SDF) model. The individual deterministic models are composed to a system model on which efficient analysis can be performed. An enabling property for ensuring the conservativeness of the analysis performed on the composite-model is the monotonicity of the individual components. Some components in CompSOC have been shown to exhibit non-monotonic behavior, hence need to be treated separately.

We propose a framework for the abstraction of non-monotonic components into SDF graphs. The abstraction is compositional in the sense that it allows for abstractions of individual components in a composite-system while preserving the conservativeness of the model. We use the framework to abstract non-monotonic components in the CompSOC communication-channel into a monotonic SDF graphs, which fits well within the CompSOC modeling framework.

To complement the modeling of the non-monotonic components of the communication-channel, we provide SDF models of all components of the channel. Our models are either new, or refinements the existing models which provide tighter analysis. Also, we develop a modeling technique that enables modeling of behavior that could not be modeled before. Namely, we model the interference of tasks running on the platform and sharing a buffer or a request-queue under static-order scheduling. This modeling technique can also be applied to general SDF modeling, outside the CompSOC framework.
# Contents

1 Introduction  
1.1 CompSOC .......................... 1  
1.2 Monotonicity ...................... 1  
1.3 CompSOC non-monotonicity ............ 2  
1.4 Models of Computation ............... 2  
1.5 Problem statement .................. 3  
1.6 Main contributions ................. 4  
1.7 Organization .................... 6  

2 Modeling  
2.1 SDF graphs ........................ 7  
2.2 Timed actor interfaces .......... 8  
  2.2.1 Discrete-Time Automata for actor representations ........................... 11  
2.3 Monotonicity .................... 12  
2.4 Performance ..................... 13  
2.5 Composition ..................... 14  
2.6 Proposed abstraction framework ..... 16  

3 CompSOC  
3.1 CompSOC architecture .............. 19  
3.2 Intertask communication protocol ... 20  
  3.2.1 Shared FIFO and counters memory allocation ................................. 22  
3.3 Programming model and platform synthesis ............ 23  
3.4 Conservative modeling of hardware resources  
  3.4.1 Processor ..................... 25  
  3.4.2 Interconnect .................. 26  
  3.4.3 Memory controller ............. 27  
  3.4.4 Model compositions ............ 27  
3.5 Shared buffer model ................ 29  
3.6 SDF execution model ............... 32  
  3.6.1 Software-level interaction with the communication-channel .......... 33  
  3.6.2 Channel top-level view ........ 35  

4 Non-monotonic abstraction ........... 38  
4.1 Data abstraction .................. 38  
4.2 claim_data: non-monotonic abstraction ..................... 38  
  4.2.1 The conditional branch actor CB<sub>n>0</sub> ...................... 39  
  4.2.2 Model ....................... 40  
  4.2.3 Non-monotonicity of claim_data ............ 42  
  4.2.4 Abstraction .................. 45  

5 Channel-Model .................... 47  
5.1 Read data ........................ 47  
5.2 claim_data and claim_space parameters ............ 48  
5.3 Writing and releasing data ............. 49  

6 Related work ..................... 55
7 Conclusions
7.1 Future work
List of Figures

1.1 The sequence of operations in the SDF programming model. ............................................. 1
1.2 Individual components composed to form an overall model of the system ......................... 2
1.3 Monotonic vs. non-monotonic illustration ........................................................................ 3
1.4 Tasks A and B communicating using a shared FIFO. ....................................................... 3
1.5 A layered representation of CompSOC ............................................................................. 4
1.6 Abstraction framework for non-monotonic components in CompSOC ............................. 5

2.1 An example SDF graph. ................................................................................................. 7
2.2 Event sequences examples ............................................................................................. 8
2.3 SDF graph abstraction examples ................................................................................... 10
2.4 Merge actor DTA description ......................................................................................... 11
2.5 Non-monotonic composition example ........................................................................... 13
2.6 Non-monotonic composition example trace .................................................................. 14
2.7 Abstraction steps: system implementation to a conservative SDF system model ............ 17

3.1 A CompSOC instance .................................................................................................. 20
3.2 C-HEAP block diagram ............................................................................................... 21
3.3 CompSOC Tool-flow .................................................................................................... 24
3.4 The tool-flow refines the channels (edges) of the application SDF graph into a binding- 
    aware channel. .............................................................................................................. 25
3.5 General dataflow model for an LR-server ..................................................................... 26
3.6 Modeling the execution of a sequence of instructions on the virtualized CPU ................ 26
3.7 NoC model ..................................................................................................................... 26
3.8 Latency-rate model for the memory. .............................................................................. 27
3.9 Writing from A to memory ............................................................................................ 28
3.10 SDF Modeling of a buffer shared among three statically-ordered tasks ....................... 30
3.11 Back-pressure edge from a shared buffer vs. back edge from a shared request-queue ...... 31
3.12 The sequence of operations in the SDF programming model. ....................................... 31
3.13 The sequence of operations in the SDF programming model with multiple incoming 
    and outgoing dependencies. ......................................................................................... 33
3.14 SDF model for the execution phases combined with channel interaction .................... 34
3.15 The communication-channel between two tasks $t_1$ and $t_2$, denoted by $(t_1,t_2)$ .... 34
3.16 Channel interface ......................................................................................................... 35
3.17 Execution illustration of a task with multiple channels ............................................... 37

4.1 The data-dependent implementation is abstraction into the event-based TAI and SDF 
    models. .......................................................................................................................... 38
4.2 Conditional-branch CB$_{n>0}$ actor ............................................................................... 39
4.3 CD actor for claim_data ................................................................................................ 41
4.4 Model for claim_space ................................................................................................. 42
4.5 The claim_data model ................................................................................................. 43
4.6 Counterexample: claim_data is not ⊑-monotonic .......................................................... 44
4.7 Intuitive illustration: claim_data is non-monotonic ...................................................... 45
4.8 SDF model for claim_data ............................................................................................ 45

5.1 SDF model for a Read operation of a data token on channel $i$ of task $C$ ..................... 48
5.2 Models used for extracting values for $\beta_{req}$, $\beta_{cmp}$ ................................................ 49
5.3 CD interaction with other SDF actors .......................................................................... 49
List of Tables

3.1 C-HEAP synchronization primitives .................................................. 21
3.2 Units summary ................................................................................. 27
3.3 Write transaction model parameters .................................................. 28
3.4 Input and output ports for CPU model of task $t$ .............................. 35
3.5 Input, output and internal ports for a channel $(l_1, t_2)$ .................... 36
4.1 Input and output ports of CD ............................................................. 40
4.2 System specific parameters used in the CD model ............................. 41
5.1 Common notations used throughout the primitive models ................. 47
Acronyms and abbreviations

\textbf{LR-server} Latency-Rate server.
\textbf{C-HEAP} CPU-controlled HEterogeneous Architectures for signal Processing.
\textbf{CCSP} Credit-Controlled Static Priority.
\textbf{CMEM-i} input Communication Memory.
\textbf{CMEM-o} output Communication Memory.
\textbf{CompSOC} COmposable and predictable Multi-Processor System On Chip.
\textbf{DMA} Direct Memory Access.
\textbf{DMEM} Data Memory.
\textbf{DTA} Discrete-Time Automata.
\textbf{IMEM} Instruction Memory.
\textbf{MoC} Model of Computation.
\textbf{MSU} Memory Service Unit.
\textbf{NI} Network Interface.
\textbf{NoC} Network on Chip.
\textbf{OS} Operating System.
\textbf{RT} Real-time.
\textbf{SDF} Synchronous Data Flow.
\textbf{TAI} Timed-Actor Interfaces.
\textbf{WC} Worst-Case.
\textbf{WCRT} Worst-Case Response-Time.
Chapter 1

Introduction

Real-time (RT) systems must adhere not only to functional requirements, but also to temporal requirements. Whereas systems classified as soft RT can tolerate occasional violations of timing requirements, firm and hard RT systems have strict timing requirements, where the latter are also safety critical [5]. To adhere to these strict temporal requirements, platforms must be predictable and therefore provide Worst-Case (WC) temporal guarantees. One such platform is the COmposable and predictable Multi-Processor System On Chip (CompSOC) [10].

1.1 CompSOC

CompSOC is a platform template that provides WC performance guarantees to its applications. The platform is not only predictable, but also composable, i.e., allows the individual applications to be developed, verified and executed in isolation. Composability is achieved by eliminating interference between applications. To achieve predictability, two measures are taken:

1. The platform resources (e.g., the processors, the interconnect, and the memories) and their composition are formally modeled using the Synchronous Data Flow (SDF) [14] Model of Computation (MoC), and

2. A user application is obliged to comply with the SDF programming model. This requires each task to operate according to the sequence of operations depicted in Figure 1.1. First, the task ensures that sufficient space is available in memory and acquires all required data for its execution, denoted by operations Claim and Read in Figure 1.1. Then the task executes, and only after execution has finished, writes any data to memory and releases any space that is now made available. The latter two are denoted by operations Write and Release. In addition to operating according to the above sequence, the amount of data required for a task to execute and the amount of data produced are fixed for all executions of the same task.

The overall system, including both the user applications and the platform resources, is modeled as a closed SDF graph for which WC performance metrics can be extracted. Each of the system components is modeled individually as an SDF graph, and the components are composed together to an SDF model of the system, as illustrated by Figure 1.2.

1.2 Monotonicity

For each component, only the WC temporal behavior is considered, instead of the entire range of possible behaviors. The motivation for considering only WC behavior is that it results in a light-weight model that can be efficiently analyzed [22]. One of the key properties enabling the validity of the analysis is the monotonicity of SDF graphs [17]. SDF graphs are monotonic in the sense that an earlier occurrence of
Component models

An event modeled in the graph cannot result in a later occurrence of another event. Figure 1.3 illustrates the potential effect of a non-monotonic component. Figures 1.3(a)-(b) illustrate the finishing-time of a non-monotonic component, and Figures 1.3(c)-(d) illustrate the effect of the component on the over-all composite-system performance.

Figures 1.3(a)-(b) compare the behavior of a monotonic component versus that of a non-monotonic one. The component executes as a result of an event arrival at its input and produces an event at its output at a time denoted by \( \text{finishing-time} \).

In Figure 1.3(a) we observe monotonic behavior: a later arrival-time always implies a later finishing-time. In contrast, the behavior illustrated in Figure 1.3(b) is non-monotonic: the later arrival of an event may imply an earlier finishing-time. Figures 1.3(c)-(d) illustrates the effect of the components on the over-all system performance. If we consider only the WC arrival-time of the input event at the input of a non-monotonic component, we may overlook the WC behavior of the composite-system, resulting in a non-conservative finishing-time.

1.3 CompSOC non-monotonicity

Intertask communication in CompSOC is performed using the CPU-controlled HEterogeneous Architectures for signal Processing (C-HEAP) protocol [16], implemented at the Operating System (OS) level. Tasks communicate over a shared FIFO, as illustrated by Figure 1.4.

Whereas the execution of application tasks and the hardware resources are monotonic, the communication protocol does not adhere to monotonic behavior [15], as it involves polling procedures. During a polling procedure, time-consuming iterations take place until a certain condition holds. If response-time for checking the condition is shortened, it is possible that more iterations are required, potentially causing a later finishing-time for the overall calculation. This non-monotonicity is thoroughly examined in Section 4.

Figure 1.5 provides a layered representation of CompSOC. At the bottom layer, the hardware resources such as CPUs, memories, and an interconnects are shown. On top of the hardware, an OS layer provides scheduling and communication services to the user applications. The non-monotonic components are part of the OS services, and are highlighted in gray. The presence of a non-monotonic component inhibits the desired compositional WC analysis of predictable behavior.

1.4 Models of Computation

Verification of requirements may be performed directly on a system. However, this is often cumbersome or even infeasible due to the complexity of the system. Furthermore, when the intention is to synthesize a system, the characteristics of the system need to be formalized as a model. For these purposes, we require a model of computation. In MoCs there is a tradeoff between \textit{expressiveness} and \textit{analizability}. Timed SDF models are used to conservatively model system implementations, thus enabling efficient system analysis. Whereas SDF models are extremely analyzable they have limited expressiveness. One
Figure 1.3: Monotonic vs. non-monotonic illustrations. The behavior at the component level is illustrated by Figures (a)-(b): considering only the finishing-time of the WC arrival-time is conservative for the monotonic component but is not conservative for the non-monotonic component, as it overlooks the dashed area in the graph. The effect at the composite-system level is illustrated by Figures (c)-(d): for a non-monotonic component, overlooking the dashed area in the graph may result in a non-conservative analysis (i.e., optimistic) of the overall system performance.

Figure 1.4: Tasks A and B communicating using a shared FIFO.

The theory of Timed-Actor Interfaces (TAI) [8] is a very general theory that describes the temporal behavior of components as a sequence of events. This theory is expressive enough to model non-monotonic components. The theory also defines a useful notion of abstraction of models, allowing the substitution of components in certain contexts, while preserving performance guarantees. Using TAI to model non-monotonic components, and using TAI’s already established theory for transitioning between formalisms seems to be a promising direction for the modeling of non-monotonic behavior.

1.5 Problem statement

The aim of this thesis is two-fold. At the basic level, our goal is to provide a model for the C-HEAP communication protocol such that:

1. It conservatively bounds (i.e., gives a safe over-estimation) the temporal behavior of the actual implementation, and
2. It is easily integrated within the existing CompSOC modeling framework.

To achieve both of these goals we aim at providing an SDF model for the components of the protocol and defining the way it should be composed with the existing CompSOC models.

At a higher level, we aim at using this case-study to understand non-monotonic behavior and explore the possibilities for conservative abstraction of non-monotonic systems in general.

1.6 Main contributions

The main contributions of this thesis are the following.

1. A framework for the abstraction of non-monotonic components into SDF models. In our framework, illustrated by Figure 1.6, we take a two-stage modeling approach. First, we model the non-monotonic components within the TAI modeling framework, where the non-monotonic behavior can be captured. In Figure 1.6, this modeling is illustrated as the transition from the implementation to a conservative TAI model. Then, we abstract the TAI model to a (monotonic, more analyzable) SDF model, illustrated as the second transition in Figure 1.6. The modeling of monotonic components can be done in a single stage process, directly modeling the implementation as a conservative SDF model (as is traditionally done for the monotonic components in CompSOC).

2. A conservative SDF model for the non-monotonic primitives of the C-HEAP communication protocol.

3. A structured channel-model and task-execution model. Our models are either additional (i.e., new), refinements (i.e., provide tighter analysis) or corrections of existing channel models.

4. A model for a shared-buffer under static-order scheduling. This model enabled the analysis of interference between statically-ordered tasks of the same application using a shared buffer or a shared request queue.
Figure 1.6: Abstraction framework for non-monotonic components in CompSOC.
1.7 Organization

The rest of this thesis is organized as follows. First, in Chapter 2 we provide the theoretical background for the modeling formalisms used in this thesis, namely SDF and TAI, and discuss our modeling framework for abstraction of non-monotonic components. Then, in Chapter 3 we provide an overview of CompSOC’s architecture and tool-flow, and explain how each of the HW resources is currently modeled. In this chapter we also contribute a model for a shared buffer under static-order scheduling, and provide a top-level view on our proposed communication-channel model. In the following two chapters we provide low-level models for the components of the communication-channel. In Chapter 4, we present a conservative model for the non-monotonic components, and in Chapter 5 we model the remaining components and construct a full channel-model. Chapter 6 contains a discussion on related work. Finally, in Chapter 7 we provide conclusions and future work.
Chapter 2

Modeling

In the CompSOC platform the system behavior is formally modeled. In this chapter we present two modeling formalisms. The first is SDF graphs, introduced in [14]. The SDF formalism is used in CompSOC for timing analyses for latency and throughput. Whereas the model is useful for performance analysis, it is limited in expressiveness and cannot directly be used to model non-monotonic behavior. The second formalism we present in this chapter is TAI [8], a formalism that is more expressive than SDF. TAI also defines an abstraction/refinement relation that could be used to aid in the integration of expressive formalisms into SDF modeling frameworks. We conclude this chapter by proposing an abstraction framework for such integration.

2.1 SDF graphs

Streaming applications [24] operate on a large a sequence of data items on which typically a set of independent filters is applied. Such applications can often be modeled as SDF graphs and analyzed for performance and buffer capacities [19]. In SDF graphs, the production of data is abstracted to tokens, where each token represents a quantum of data. The components of the system are called actors and the dependencies between them are represented as channels transferring tokens. A channel represents dependency between two actors.

Figure 2.1: An example SDF graph.

Figure 2.1 provides an example of an SDF graph with three actors: a, b, and c. The numbers inside the actor nodes denote their execution-times. The edges in the graph are the channels. Channels connect actors via ports. Next, we define actors and channel more formally. We assume a set Ports of ports.

Definition 1 (SDF actor). An SDF actor is a 3-tuple \((P, Q, d)\) where

- \(P \subseteq \text{Ports}\) is the set of input ports,
- \(Q \subseteq \text{Ports}\) is the set of output ports,
- \(P \cap Q = \emptyset\), and
- \(d \in \mathbb{R}^{\geq 0}\) is the delay of the actor.

Definition 2 (SDF channel). An SDF channel is a 3-tuple \((\text{out}, \text{in}, \text{init})\) where

- \(\text{out}, \text{in} \in \text{Ports}\), and
- \(\text{init}\) is the number of initial tokens available on the channel.

Each port has a rate associated to it.
Definition 3 (Rate). Rate : Ports → \( \mathbb{N}^+ \) is a mapping from ports to the positive numbers.

The execution of an SDF actor is called a firing. In each firing, on each input port \( i \), a fixed number of Rate\((i) \) tokens is consumed from the channel connected to \( i \). Similarly, for each output port \( o \), a fixed number of Rate\((o) \) tokens is produced. As illustrated in Figure 2.1, the rates are the numbers appearing next to the source of the channel in the case of an input port, and next to the destination in the case of an output port. If the channel has an initial number of tokens, it is denoted by a black dot on the edge. The number appearing next to the dot is the number of initial tokens. For example, in Figure 2.1, the channel between actor \( b \) and \( c \) has 5 initial tokens, but the channel from \( c \) to \( b \) has none.

The (self-timed) execution of the actor starts as soon as there are enough tokens available on its inputs. The condition for firing is called the firing rule. The execution of the actor lasts \( d \) time units. The tokens at the output will be produced as soon as the execution of the actor has finished: if there are sufficiently many tokens at the inputs at time \( t_0 \), and the execution-time of the actor is \( d \), the tokens will be available at the output ports at time \( t_0 + d \).

SDF actors and channels compose an SDF graph.

Definition 4 (SDF graph). An SDF graph is a tuple \((A, C)\) with a finite set \( A \) of actors and a finite set \( C \subseteq \text{Ports}^2 \) of channels. The source of every channel is an output port of some actor; the destination is an input port of some actor. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actor.

Figure 2.1 provides an example of an SDF graph which is a composition of the actors \( a, b, \) and \( c \). Notice that some edges (e.g., the channel between \( a \) and \( b \)) do not explicitly state the rates of some ports. This denotes that the rate of the port is implicitly 1. Similarly, if a black dots is not augmented with a number, the number of initial tokens is implicitly 1. Also notice that the actor \( a \) has a self-edge with a single initial token. As such, the actor can only execute sequentially, i.e., actor firings can only take place one at a time.

2.2 Timed actor interfaces

Formal analysis of SDF graphs relies on their monotonicity. However, as we observe in Chapter 5, some components in CompSOC behave non-monotonically. In this section we provide the theory of Timed Actor Interfaces (TAI), introduced in [8]. The theory is a generalization of SDF and can be used to provide conservative models for the execution of non-monotonic components. In Section 2.6 we explain how TAI models can be integrated into SDF models using the TAI notion of abstraction.

Similar to SDF graphs, TAI actors abstract away from data values, and an event is represented as a token. Each token has a time-stamp in the time domain. We use the positive real numbers \( \mathbb{R}^{\geq 0} \) with explicitly adding the maximal element \( \infty \) as our continuous time domain. We denote our time domain as \( \mathcal{T}^{\infty} \).

Definition 5 (Event sequences). An event sequence is a total mapping \( \tau : \mathbb{N} \rightarrow \mathcal{T}^{\infty} \), such that \( \tau \) is weakly monotonic, that is for every \( k, m \in \mathbb{N} \) and \( k \leq m \), we have that \( \tau(k) \leq \tau(m) \).

For an event sequence \( \tau \), the length of the event sequence is the smallest \( n \in \mathbb{N} \) such that \( \tau(n) = \infty \), denoted by \( |\tau| \). An event sequence can either be finite or infinite. In the latter case \( |\tau| = \infty \). In case \( \tau \) is finite, every event index \( n \geq |\tau| \) is mapped to \( \infty \), so \( \forall n \geq |\tau|, \tau(n) = \infty \). We use \( \epsilon \) to denote the empty event sequence, where \( \forall n : \tau(n) = \infty \). Event sequences can be explicitly expressed as a concatenation of time-stamps, separated by dots. For example, the event sequence consisting of three events at the times 2, 4, and 7 is expressed as \( 2 \cdot 4 \cdot 7 \cdot \epsilon \), as illustrated by \( \tau_1 \) in Figure 2.2. The set of all event sequences is \( \mathcal{T}r \).

![Event sequences](image)

Figure 2.2: Event sequences \( \tau_1 = 2 \cdot 4 \cdot 7 \cdot \epsilon \), \( \tau_2 = 1 \cdot 4 \cdot 6 \cdot \epsilon \), and \( \tau_3 = 1 \cdot 4 \cdot 6 \cdot 12 \cdot \epsilon \)
Event sequences are communicated over ports. For a set P of ports, Tr(P) = \{f : P \rightarrow Tr \mid f \text{ is total}\} is the set of total functions of type P \rightarrow Tr that map each port in P to an event sequence in Tr. Elements of Tr(P) are called event traces over P.

**Definition 6** (Earlier-than and prefix orders). For two event sequences \( \tau, \tau' \), the event sequence \( \tau \) is said to be earlier than \( \tau' \), denoted by \( \tau \leq \tau' \), iff

1. \(|\tau| = |\tau'| \), and
2. \( \forall n < |\tau|, \tau(n) \leq \tau'(n) \).

The event sequence \( \tau \) is said to be a prefix of \( \tau' \), denoted by \( \tau \preceq \tau' \), iff

1. \(|\tau| < |\tau'| \), and
2. \( \forall n < |\tau|, \tau(n) = \tau'(n) \).

\( \leq \) and \( \preceq \) are lifted to event traces \( x, x' \in Tr(P) \) in the standard way: \( x \leq x' \) iff for all \( p \in Tr(P) \), \( x(p) \leq x'(p) \); \( x \preceq x' \) iff for all \( p \in Tr(P) \), \( x(p) \preceq x'(p) \).

**Definition 7** (Abstraction and refinement of event sequences and event traces). An event sequence \( \tau' \) abstracts and event sequence \( \tau \) (\( \tau \) refines \( \tau' \)), denoted by \( \tau \sqsubseteq \tau' \) iff \( \forall n \in \mathbb{N}, \tau(n) \leq \tau'(n) \). \( \sqsubseteq \) is lifted to event traces in the standard way: \( x, x' \in Tr(P) \) : \( x \sqsubseteq x' \) iff for all \( p \in P \), \( x(p) \sqsubseteq x'(p) \).

In the example traces of Figure 2.2, \( \tau_2 \leq \tau_1 \), and \( \tau_2 \sqsubseteq \tau_1 \). Notice that also in the general case, for two sequences \( \tau, \tau' \in Tr \), it always holds that \( \tau \leq \tau' \Rightarrow \tau \preceq \tau' \), and also by definition \( \tau \preceq \tau' \Leftrightarrow \tau \sqsubseteq \tau' \wedge |\tau| = |\tau'| \). Furthermore in Figure 2.2, \( \tau_1 \subseteq \tau_1 \), but not \( \tau_2 \subseteq \tau_1 \), because \( |\tau_2| = |\tau_1| \). More in Figure 2.2, \( \tau_2 \sqsubseteq \tau_3 \) and \( \tau_3 \sqsubseteq \tau_2 \). Also in the general case, \( \tau \preceq \tau' \Rightarrow \tau \sqsubseteq \tau' \).

**Definition 8** (Actors). An actor \( A \) is a 3-tuple \( A = (P, Q, R_A) \), where:

- \( P \) is the set of input ports,
- \( Q \) is the set of output ports, and
- \( R_A \) is the event trace relation \( R_A \subseteq Tr(P) \times Tr(Q) \) between the input traces and output traces of actor \( A \).

We will sometimes use \( xAy \) to denote \((x, y) \in R_A\).

**Definition 9** (Legal input traces). Let \( A = (P, Q, R_A) \) be an actor. \( in_A \) denotes the set of all legal input traces of \( A \): \( in_A = \{x \in Tr(P) \mid \exists y \in Tr(Q) : xAy\} \).

An actor \( A \) with a set \( P \) of input ports is called input-complete iff \( in_A = Tr(P) \).

**Definition 10** (Abstraction and refinement of actors). Let \( A = (P, Q, R_A) \) and \( A' = (P, Q, R_{A'}) \) be two actors. The actor \( A' \) abstracts the actor \( A \) (\( A \) refines \( A' \)), denoted by \( A \sqsubseteq A' \) iff

1. \( in_{A'} \subseteq in_A \), and
2. \( \forall x \in in_{A'}, \forall y \in Tr(Q) : xAy \Rightarrow \exists y' \in Tr(Q) : y \sqsubseteq y' \wedge xA'y' \).

It is useful to note that \( \sqsubseteq \) for actors is a preorder, i.e., it is reflexive and transitive.

In Figures 2.3(a)-2.3(c), we provide several examples of abstraction applied to SDF graphs with a single input port \( p \) and a single output port \( q \). The SDF actors of Figure 2.3(a) differ in the delay of their execution. For any input event trace \( x \) and the output event traces \( y, y' \), such that \( xAy \) and \( xA'y' \), it holds that \( y \sqsubseteq y' \), as \( |y| = |y'| \), and \( \forall n < |y|, y(n) = y'(n) + 5 \). For the same input trace, the output tokens of \( A' \) will always be produced later than by \( A \). In the example in Figure 2.3(b), the actors have the same execution delay, but the actor \( B' \) has an additional dependency edge. This self-edge requires that an execution of actor \( B' \) must finish before another execution can start. This will result in a sparse output trace, where all tokens have a difference of at least 5 time units between them. This is not the case for actor \( B \), where tokens can be executed in parallel. For a given input event trace, the output tokens on actor \( B' \) will be produced no-earlier-than the tokens at the output trace of actor \( B \). In the example of Figure 2.3(c), the delays of the actors are the same, but the actor \( C \) produces more tokens in each firing. For any input trace, the actor \( C \) will produce no-fewer tokens than actor \( C' \), and the tokens will be produced no-later-than the tokens on \( C \).
\(p \rightarrow A \quad \quad \quad \quad p \rightarrow A'\)

(a)

\(p \rightarrow B \quad \quad \quad \quad p \rightarrow B'\)

(b)

\(p \rightarrow C \quad \quad \quad \quad p \rightarrow C'\)

(c)

Figure 2.3: Abstraction examples on SDF graphs.

**Lemma 1** (\(\preceq\), \(\preceq\), and \(\sqsubseteq\) are partial orders([8])). The relations \(\preceq\), \(\preceq\), and \(\sqsubseteq\) on event sequences and event traces are partial orders, i.e., reflexive, transitive and antisymmetric.

We use \(\sqsubseteq\) to denote a partial order (e.g., \(\leq\), \(\preceq\), or \(\sqsubseteq\)).

**Definition 11** (Monotonicity and continuity). Let \(A = (P, Q, R_A)\) be an actor. Given a partial order \(\sqsubseteq\) on \(\text{Tr}(P)\) and \(\text{Tr}(Q)\), \(A\) is called \(\preceq\)-monotonic iff for every \(x, y, \) and \(x'\) such that \(xAy\), \(x' \in \text{in}_A\), and \(x \preceq x'\), there exists a \(y'\) such that \(y \preceq y'\) and \(x'y'\). Assuming \(\sqsubseteq\) yields pre-CPOs, \(A\) is \(\sqsubseteq\)-continuous iff for every pair \(\{x_k\}\) and \(\{y_k\}\) of chains w.r.t \((\text{Tr}(P), \sqsubseteq)\) and \((\text{Tr}(Q), \sqsubseteq)\) respectively, if for all \(k (x_k, y_k) \in R_A\), then the least upper bound of \(\{x_k\}\) is related to the least upper bound of the \(\{y_k\}\) : 

\[
\bigcup_{\sqsubseteq} \{x_k\}, \bigcup_{\sqsubseteq} \{y_k\} \in R_A.
\]

**Definition 12** (Determinism). An actor \(A = (P, Q, R_A)\) called deterministic if \(R_A\) is a partial function.

Notice that for deterministic actors, continuity implies monotonicity [7]. SDF graphs are \(\sqsubseteq\)-continuous, \(\leq\)-continuous and \(\sqsubseteq\)-continuous. They are also deterministic, hence monotonic in all of the above relations.

In the remainder of this chapter, we provide examples of actors. These actors are taken from [8] and will act as building blocks of the composite model that will be presented in Section 4 and 5.

**Example 1** (The variable delay actor \(\Delta_{[d_{\text{min}}, d_{\text{max}}]}\)). The variable delay actor has a single input port \(p\), a single output \(q\), and produces a single output event for each input event. For the \(n\)-th input event, the \(n\)-th output event is generated with a delay in the range \([d_{\text{min}}, d_{\text{max}}]\). The relation between the input and output event sequences is then as follows:

\[
x \Delta_{[d_{\text{min}}, d_{\text{max}}]} y \iff |x(p)| = |y(q)| \land \forall n < |x(p)| : x(p)(n) + d_{\text{min}} \leq y(q)(n) \leq x(p)(n) + d_{\text{max}}
\]

\[
n > 0 \Rightarrow y(q)(n) \geq y(q)(n - 1)
\]

The variable delay actor is \(\leq\)-monotonic, and is non-deterministic. Notice that the non-determinism of the actor is restricted to the timing of the generated events at the output \(q\). However, the number of events to be generated for a given input trace is deterministic.

**Example 2** (The merge actor \(G\)). A merge actor \(G\) is an actor with the input ports \(p_1, p_2\) and an output port \(q\) such that:

\[
xG y \iff y(q) = x(p_1)x(p_2)
\]

where the merge operator \(|\) for two event sequences is inductively defined as follows:
\[ \tau_1 \tau_2 = \begin{cases} \tau_2 & \text{if } \tau_1 = \epsilon \\ \tau_1 & \text{if } \tau_2 = \epsilon \\ t_1 \cdot (\tau_1 | \tau_2) & \text{if } \tau_1 = t_1 \cdot \tau_1', \tau_2 = t_2 \cdot \tau_2', t_1 \leq t_2 \\ t_2 \cdot (\tau_1 | \tau_2') & \text{if } \tau_1 = t_1 \cdot \tau_1', \tau_2 = t_2 \cdot \tau_2', t_2 \leq t_1 \end{cases} \]

The merge actor is $\leq$-monotonic and deterministic. Whereas the actors above were described by defining the explicit relation between their input and output sequences, it is also possible to implicitly describe the relation as automata, as explained next.

### 2.2.1 Discrete-Time Automata for actor representations

The relation between the input and output sequences of an actor can often be naturally described as an automaton. We use a discrete-time automaton Discrete-Time Automata (DTA), a timed automaton [2] with a single clock in the discrete time domain $T = \mathbb{N}$. This automaton for actor descriptions is also used in [8], where several algorithms are provided for checking properties of actors.

**Definition 13 (Discrete-time automata).** A discrete-time automaton is a tuple $M = (S, \Sigma, \Delta, I, F)$, where

- $S$ is a set of states.
- $\Sigma$ is a set of labels, also called an alphabet.
- $\Delta : S \times (\Sigma \cup \{t\}) \times S$ is a transition relation.
- $I \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of accepting states.

An automaton $M$ has Büchi accepting states. Every accepting run of the automaton generates a word $w$ in the language of $M$, denoted by $L(M)$.

A DTA $M(A)$ can define an actor $A$ as follows. A single clock is associated with the execution of a DTA. The transitions of an automaton are labeled with elements from the set $P \cup Q \cup \{t\}$, the ports of the actor united with an explicit tick label $t$, denoting the increase of the clock by one time unit. In an initial state, the value of the clock is 0. Every word can be mapped to a unique event trace pair $Tr(w) \in Tr(P) \times Tr(Q)$. The relation of the actor is the set of all event trace pairs accepted by the language of $M$: $\{Tr(w) | w \in L(M)\}$.

Recall the merge actor of Example 2, described as an explicit relation between input and output event traces. We next define the actor using a DTA.

**Example 3 (The merge actor $G$ defined as a DTA).** The port definitions of the actor, depicted in Figure 2.4(a), are the same as in Example 2. The DTA describing the relation of the actor is depicted in Figure 2.4(b).

![Figure 2.4: Merge actor DTA description](image)

The labels of the DTA are in the set $\{p_1, p_2, q, t\}$. The DTA has a single accepting state, which is also the initial state. The accepted words of the DTA describe the relation of the merge actor.
2.3 Monotonicity

In Sections 2.2 and 2.1 we provided two formalisms, where the second is a generalization of the other. In models of computation there is a tradeoff between expressiveness and analyzability. The theory of actor interfaces is a modeling framework that very generally expresses the behavior of an actor as a binary relation between event sequences on its inputs and outputs. In this section we explain how we use the expressiveness of TAI actors, and the TAI notion of abstraction in a framework for abstraction of components into SDF graphs while preserving WC performance metrics.

In CompSOC, the objective of formal modeling and analysis of the system is to derive timing guarantees and thus achieve predictability. Typically, a system is a composition of components adhering to certain behavior. If the system components are predictable, such timing guarantees can be achieved by considering all possible behaviors of the system, i.e., its state-space. Such state-space exploration could be cumbersome, and in some cases impractical. However, considering the entire state-space is not always required to achieve WC guarantees. In [8], several useful propositions are provided to guarantee performance bounds for compositions of actors. For input-complete and monotonic actors, lower bounds on throughput and upper bounds on latency are preserved under refinement, i.e., WC performance analysis performed on an abstract actor is valid also for its refinement. This property implies that for any system component with a range of possible execution-times, it is sufficient to model the component using solely its WC execution-time.

This justification is used when modeling monotonic systems using the SDF formalism, which is extensively used in the CompSOC platform, as mentioned in Section 3.4. Instead of considering all possible behaviors of the system, an SDF abstraction considers only the WC execution-times of the system components. As an SDF actor is monotonic and input-complete, WC guarantees derived for the SDF model are valid also for the real system, which is a refinement of the model.

When an actor composition holds non-monotonic behavior (e.g., the elaborated example of Section 4.2.3), the earlier arrival of an event may cause the later arrival of another event. Models using only WC execution-times (e.g., SDF models) can no longer provide valid abstractions of the composite behavior, as illustrated by the following somewhat engineered contrived.

Example 4 (Abstraction under composition with non-monotonic actors). Consider the following two actors $A$ and $B$. Actor $A$ has a single input port $p_a$, a single output port $q_a$ and emits output tokens according to the following recursive definition:

\[
q_a(0) = p_a(0)
\]

\[
q_a(n + 1) = \begin{cases} 
10(n + 1) + p_a(n + 1) & \text{if } p_a(n + 1) - p_a(n) \geq 2 \\
10(n + 1) + p_a(n + 1) + 5 & \text{otherwise}
\end{cases}
\]

Actor $A$ samples the difference between two consecutive tokens and generates tokens accordingly. For the $(n + 1)$-th token at the input, if the delay between the current token and the previous one is larger than 2 time units, the token will be delayed by $10(n + 1)$ time units. If the delay measured from the previous token is less than 2, then the token is delayed by $10(n + 1) + 5$ time units. The actor is deterministic and input-complete. For the input sequences $t = (0 \cdot 0 \cdot e)$ and $t' = (0 \cdot 2 \cdot e)$ it holds that $t \nsubseteq t'$. The input sequence $t$ is related to the output sequence $s = (0 \cdot 15 \cdot e)$, and $t'$ is related to $s' = (0 \cdot 12 \cdot e)$. As $s \nsubseteq s'$, we can conclude that the actor $A$ is not monotonic.

Actor $B$ is a sequential variation on the variable delay actor of Example 1. The actor has a single input $p_b$ and a single output $q_b$, accepting tokens at the input sequentially, i.e., beginning execution as soon as a token is available at the input and the previous execution has ended. After execution has started, each token is delayed by a delay of $[0, 2]$ time units. More formally, the actor relates two sequences $x$ and $y$ iff the following holds:

\[
x(p_b)(0) \leq y(q_b)(0) \leq x(p_b)(0) + 2
\]

\[
\max(y(q_b)(n), x(p_b)(n + 1)) \leq y(q_b)(n + 1) \leq \max(y(q_b)(n), x(p_b)(n + 1)) + 2
\]

The actor $B$ is $\subseteq$-monotonic.

To show that abstraction under a composition with non-monotonic components does not necessarily provide WC end-to-end time bounds, we compose the actors sequentially and abstract the actor $B$ to $B'$, as illustrated in Figures 2.5(a)-2.5(b). The actor $B'$ is an SDF actor with a self-edge and an execution-time 2. It holds that $B \nsubseteq B'$.

Consider the input sequence illustrated by Figure 2.6(a), consisting of two tokens on $p$ at time 0.

For the model using the abstraction $B'$, which is deterministic, the WC output is the sequence $(2 \cdot 14 \cdot e)$ (Figure 2.6(a)). For the refined model it is easy to see that the same simple input trace exists a worse (later) output.
sequences. Figure 2.6(a) shows the latest output trace: 
\[(2 \cdot (19 - \delta)) \cdot \epsilon\], where \(\delta\) is an infinitesimally small constant. This WC behavior of the refinement can no longer be observed in the abstract model.

The example shows that abstraction of actors in a non-monotonic environment may hide the actual WC behavior of the system. In the example, we abstracted the actor \(B\) and left the original actor \(A\) in the composition. Notice that even if we do substitute actor \(A\) by an actor \(A'\) such that \(A \sqsubseteq A'\), the abstraction still does not provide any guarantees on the WC behavior, as any actor is an abstraction of itself (\(A \sqsubseteq A\)). Clearly, substituting \(A\) with itself still validates the results shown in the example.

### 2.4 Performance

When verifying performance, we are interested in the latency and throughput metrics, where latency is the delay between two events, and throughput is the average number of events per time unit. Often we care about the throughput of a system after some initial transient phase. In this case, we talk about the throughput at steady state. Then we are interested in the throughput as the limit behavior of the system. Throughout this work we focus on WC performance guarantees, i.e., the maximal latency and the minimal throughput.

In [8] the authors show that the notion of refinement of Definition 7 is strong enough to preserve WC guarantees. The WC throughput is the greatest lower bound on the throughput of an event sequence. Let \(\tau\) be an event sequence. Recall that an event sequence \(\tau\) is a discrete series of events. As such, \(T\) is a lower bound on the \(\tau\)'s throughput if \(\exists K > 0\) such that for all \(n > K\) it holds that the average number of events appearing in the sequence per time unit is greater than \(T\), i.e., \(\frac{n}{\tau(n)} > T\). As such, the greatest lower bound on the throughput of a sequence is defined as follows.

**Definition 14** (Event sequence throughput greatest lower bound). Given an infinite sequence \(\tau\), its greatest lower bound on throughput is
\[
T^{lb}(\tau) = \sup\{T \in \mathbb{R}^\geq 0 \mid \exists K > 0 : \forall n > K : \frac{n}{\tau(n)} > T\}
\]

The performance bounds of an actor are defined per input sequence.

**Definition 15** (Actor throughput greatest lower bound). Given an actor \(A = (P, Q, R_A)\), a port \(q \in Q\), and an input trace \(x \in \text{Tr}(P)\), the greatest lower bound on the throughput of \(A\) is defined as follows:
\[
T^{lb}(A, x, q) = \inf\{T^{lb}(\tau) \mid \exists y : x A y \land \tau = y(q)\}
\]

The throughput at all output ports is defined simultaneously as a vector indexed by output ports:
\[
T^{lb}(A, x) = (T^{lb}(A, x, q))_{q \in Q}
\]

For two actors \(A = (P, Q, R_A)\) and \(B = (P, Q, R_B)\), we denote that for all \(q \in Q\) it holds that \(T^{lb}(A, x, q) \leq T^{lb}(B, x, q)\) by writing \(T^{lb}(A, x) \leq T^{lb}(B, x)\). The following proposition states that notion of abstraction is strong enough to preserve WC performance.

Figure 2.5: Composition of the actors.
Figure 2.6: Example trace showing that abstraction of an actor in a non-monotonic context does not preserve WC performance bounds.

**Proposition 1 ([8]).** Let $A$ and $B$ be two actors such that $B \subseteq A$. Then for any $x \in \text{in}_A$ it holds that $T^b(A, x) \leq T^b(B, x)$

In the above definitions we focus on minimal throughput. A formal definition of maximal latency and a proof for its preservation exist in [8].

2.5 Composition

We want to make use of Proposition 1 in a composite model. It is desirable to treat the individual components of a system separately, in order to make a modular model of the system, and also in order to understand the individual performance and bottlenecks. As such we want to be able to compose components (or their abstractions) together and conclude about the performance of the composite model. Several useful propositions regarding the composition of actors are provided in [8]. Let $A = (P_A, Q_A, R_A)$ and $B = (P_B, Q_B, R_B)$ be two actors such that $P_A \cap P_B = \emptyset$ and $Q_A \cap Q_B = \emptyset$. The following three compositions allow us to express any practical composite system:

1. Parallel composition. Composes $A$ and $B$ side-by-side without interaction, as illustrated by Figure 2.7(a). Parallel composition is denoted by $A || B$.

2. Sequential composition. Composes $A$ and $B$ by connecting outputs of $A$ to inputs of $B$, as illustrated by Figure 2.7(b). The connection between output ports of $A$ and input ports of $B$ is specified by a bijection $\theta : Q_A \rightarrow P_B$. The sequential composition is then denoted by $A \theta B$.

3. Feedback composition. Feedback composition of an actor $A$ on an input $p \in P$ and an output port $q \in Q$ is achieved by connecting $q$ to $p$, denoted by $A(p = q)$.

In addition to the three compositions, we also define a hiding operation, which can be used to make internal event sequences unobservable.

For actors holding certain properties, the refinement of an individual actor implies also the refinement of the composition, as established by the following propositions. For compactness reasons, we
Proposition 2 ([8]). Let \( A, A', B, \) and \( B' \) be input-complete actors such that \( A' \subseteq A \) and \( B' \subseteq B \). Then:

1. \( A \theta B' \subseteq A \theta B \)

2. if \( B \) is \( \sqsubseteq \)-monotone, then \( A' \theta B \subseteq A \theta B \)

Proposition 3 ([8]). Let \( A \) and \( A' \) be input-complete actors such that \( A \) is \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous, \( A' \) is \( \preceq\)-monotone and \( \preceq\)-continuous, and \( A' \subseteq A \). Then \( A'(p = q) \subseteq A(p = q) \).

If the refined component is input-complete, \( \preceq\)-monotone and \( \preceq\)-continuous, and if the abstract component is input-complete, \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous, then any performance guarantees retrieved for the composition of the abstract components also holds for the composition of the refined components.

Since parallel composition involves no interaction between the components in the composition, any analysis performed on abstractions of actors involved in the composition also holds for their refinement.

Propositions 2 and 3 state that the abstraction relations hold also on compositions of abstracted components under certain conditions. Namely, the refined actors need to be \( \preceq\)-monotone and \( \preceq\)-continuous and the abstracted actors need to be \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous. We want to be able to apply compositions multiple times when creating a composite model.

Lemma 2. Let \( A \) and \( B \) be two input-complete actors. Then:

1. If \( A \) and \( B \) are \( \preceq\)-monotone and \( \preceq\)-continuous, then also \( A \theta B \) is input-complete, \( \preceq\)-monotone and \( \preceq\)-continuous.

2. If \( A \) and \( B \) are \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous, then also \( A \theta B \) is input-complete, \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous.

3. If \( A \) is \( \preceq\)-monotone and \( \preceq\)-continuous, then \( A(p = q) \) is input-complete, \( \preceq\)-monotone and \( \preceq\)-continuous([8]).

4. If \( A \) is \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous, then \( A(p = q) \) is input-complete, \( \sqsubseteq\)-monotone and \( \sqsubseteq\)-continuous.

Proof.

1. (input-completeness) \( A \) is input-complete, and as such also \( A \theta B \).

(\( \preceq\)-monotonicity) We need to show that for all \( x_a, x_b, y_a \) such that \( x_a \leq x_b \) and \( x_a \theta B y_a \), there exists a \( y_b \) such that \( y_a \leq y_b \) and \( x_b \theta y_b \). Let \( z_a \) be and output traces of \( A \) such that \( x_a z_a \). Because of the \( \preceq\)-monotonicity and input-completeness of \( A \) we know that there exists a \( z_b \) such that \( x_b z_b \) and \( z_a \leq z_b \). Let \( z^\prime \) be a trace of \( P_B \), and let \( z^\prime_a \) and \( z^\prime_b \) be traces on the inputs of \( B \) such that all ports of \( z \) defined by the bijection \( \theta \) are replaced by \( \theta(z_a) \) in \( z^\prime_a \) and by \( \theta(z_b) \) in \( z^\prime_b \). Since \( z_a \leq z_b \), we know that also \( z^\prime_a \leq z^\prime_b \). Because \( B \) is input-complete and monotonic, there exists a \( y_a, y_b \) such that \( z^\prime_b B y_a, z^\prime_b B y_b \) and \( y_a \leq y_b \). By construction it holds that \( x_a \theta B y_a \) and \( x_b \theta B y_b \) and we have
showed that \( AB \theta B \) is \( \preceq \)-monotone.

(\( \preceq \)-continuity) Because \( A \) is \( \preceq \)-continuous, for any two chains \( \{ x_k \} \) and \( \{ z_k \} \) it holds that
\[
\bigcup_{k \in \mathbb{N}} \{ x_k \} \bigcup_{k \in \mathbb{N}} \{ z_k \}.
\]
From any chain \( \{ w_k \} \) we can construct a chain \( \{ z'_{\theta} \} \) by replacing for each \( w_k \) all ports of \( w_k \) defined by the bijection \( \theta \) by \( \theta(z_k) \). Since \( B \) is \( \preceq \)-continuous, we know that for any chain \( \{ y_k \} \) it holds that \( \bigcup_{k \in \mathbb{N}} \{ z'_{\theta} \} \bigcup_{k \in \mathbb{N}} \{ y_k \} \). And as such by construction it holds that for any chain \( \{ x_k \} \) and \( \{ y_k \} \) it holds that \( \bigcup_{k \in \mathbb{N}} \{ x_k \} \bigcup_{k \in \mathbb{N}} \{ y_k \} \).

2. The proof is almost identical to the proof of the previous item (1), with \( \preceq \) replaced by \( \sqsubseteq \), hence we do not provide it here.

3. (8)

4. To show that \( \sqsubseteq \)-monotonicity and \( \preceq \)-continuity are preserved under feedback, we use similar argumentation to that presented in [8] in the proof of the previous item (3). Let \( A = (P, Q, R) \), with \( p \in P \) and \( q \in Q \). If \( x \) is an event trace then \( x[p \to \tau] \) denotes the event trace obtained from \( x \) by setting \( p \) to \( \tau \) and leaving the other port sequences in \( x \) unchanged.

(input-completeness) We show that for all \( x \in Tr(P \setminus \{ p \}) \), it holds that \( x \in in_{A(p=q)} \). Let \( x \in Tr(P \setminus \{ p \}) \). Define \( x_0 \) and \( x_0' \) such that \( x_0 = x[p \to \epsilon] \in in_A \). Hence there exists \( y_0 \) such that \( x_0 \uparrow y_0 \). We define \( x_1 \) as follows: \( x_1 = x[p \to y_0(q)] \). Since \( A \) is \( \sqsubseteq \)-monotone, it holds that \( x_0 \sqsubseteq x_1 \). Furthermore, by \( \preceq \)-continuity, there exists a \( y_1 \) such that \( x_1 \downarrow y_1 \) and \( y_1 \not\sqsubseteq y_1 \). We repeat these iterations with \( x_{k+1} = x[p \to y_k(q)] \) to create two chains \( \{ x_k \} \) and \( \{ y_k \} \) in the \( \sqsubseteq \)-CPO, such that for all \( k \), \( x_k \downarrow y_k \). Let \( x' = \bigcup_{k \in \mathbb{N}} \{ x_k \} \) and \( y' = \bigcup_{k \in \mathbb{N}} \{ y_k \} \). Then by construction and by \( \preceq \)-continuity \( \[x'(p) = y'(p) \] \) and \( x' \downarrow y' \). We have constructed an \( x' \) such that \( x' \uparrow \{ p \} = x \). This can be done for any \( x \) and so \( A(p=q) \) is input-complete.

(\( \sqsubseteq \)-monotonicity) Let \( x_a, x_b \in Tr(P \setminus \{ p \}) \), such that \( x_a \sqsubseteq x_b \) and \( x_a' \in Tr(P) \), \( y_a' \in Tr \) such that \( x_a' \uparrow \{ p \} = x_a \) and \( x_a'(p) = y_a(q) \). In other words, \( x_a, A(p=q)y_a \). We show that \( A(p=q) \) is \( \sqsubseteq \)-monotone by showing that there exists a sequence \( y_b' \) such that \( x_b, A(p=q)y_b' \). Let \( x_b' = x_b[p \to y_b(q)] \). Then \( x_b' \sqsubseteq x_b \). By \( \preceq \)-monotonicity and input-completeness of \( A \), there exists \( y_b' \) such that \( x_b, A(p=q)y_b' \) and \( y_b' \not\sqsubseteq y_b' \). We repeat the construction as follows. For \( k > 0 \), let \( x_{b,k} = x_{b,k-1}[p \to y_{b,k-1}(q)] \). Then \( x_{b,k-1} \sqsubseteq x_{b,k} \). Since \( A \) is \( \sqsubseteq \)-monotone and input-complete, we always proceed with this iterative procedure, and there exists a \( y_{b,k} \) such that \( x_{b,k}, Ay_{b,k} \) and \( y_{b,k} \not\sqsubseteq y_{b,k} \). It follows inductively that for all \( k \), \( x_{b,k} \uparrow \{ p \} = x_b \) and \( y_{b,k} \uparrow \{ y_{b,k}(p) \) and \( y_{b,k} \not\sqsubseteq y_{b,k} \). As such we can conclude that \( x_b, A(p=q)y_{b} \) and \( A(p=q) \) is \( \sqsubseteq \)-monotone.

(\( \preceq \)-continuity) Assume we have two chains of inputs and outputs of \( A(p=q) \), \( x_a \) and \( y_k \) respectively, such that \( x_a, A(p=q)y_k \). Then there exist the chains \( x_1 \) such that for all \( k \), \( x_k = x_1 \uparrow \), \( x_1, Ay_{b,k} \), with \( x_1(p) = y_k(q) \). From this it follows that also \( x_k \) is a chain in the \( \sqsubseteq \)-CPO. Since \( A \) is \( \sqsubseteq \)-continuous, it holds that for the sequences \( \{ x_k \} \) and \( \{ y_k \} \), the lower upper bounds are in \( RA \):
\[
\bigcup_{k \in \mathbb{N}} \{ x_k \} \bigcup_{k \in \mathbb{N}} \{ y_k \}.\]

2.6 Proposed abstraction framework

The motivation behind the abstraction framework is to individually handle the system components, which is desirable for several reasons. First, separate analysis makes the system modular, i.e., allows for individual components to be replaced by other components holding similar functionality. In addition, modularity allows for easier instantiation of multiple instances of the same component. This property becomes especially important in the case of a synthesized system with synthesized models (e.g., CompSOC). Furthermore, handling smaller components individually decreases the complexity of the modeled object, and thus increases our confidence in the correctness of the model. Provided that certain conditions are met (which we review next), the TAI notion of abstraction/refinement preserves the WC performance guarantees of a composite-model under abstraction of its individual components. Our method for abstraction is illustrated by Figure 2.7.

First, we describe the individual components as TAI models. If a conservative SDF model can easily be extracted for a component (e.g., a sequential execution of instructions with no external interaction), or if an SDF model already exists for the components, then we opt for an SDF model (which is also a TAI model). We must ensure that our models are \( \preceq \)-monotone and \( \preceq \)-continuous (for SDF this is already
true). In this work we provide examples of several actors that can be used for modeling. However, we do not wish to restrict the TAI models to a specific private formalism. If we are not able to describe the components of the system as a \( \leq \)-monotone, \( \leq \)-continuous TAI models, then abstraction needs to be proven separately and cannot be based on the justification presented in this section. In the test-case studied in this work, we were able to model the behavior of the component using a \( \leq \)-monotone, \( \leq \)-continuous TAI model. Classification of systems which can or cannot be modeled as such TAI models is an interesting open question that is left for future work.

![Diagram](attachment:image.png)

Figure 2.7: Abstraction steps: system implementation to a conservative SDF system model. Each of the system components \( C_i \) is treated individually and modeled as TAI actors \( A'_i \). Then, each of the TAI actors is abstracted into a corresponding SDF actor \( A_i \). The SDF actors are composed together to model the original system.

Once we have \( \leq \)-monotone and \( \leq \)-continuous TAI models of the individual components, we abstract these into SDF models, which are input-complete, \( \sqsubseteq \)-monotone and \( \sqsubseteq \)-continuous.

Finally, we compose the SDF models together in the same manner as the original system. Since the original model is \( \leq \)-monotone and \( \leq \)-continuous, and since SDF models are \( \sqsubseteq \)-monotone and \( \sqsubseteq \)-continuous, and since these properties are preserved by composition (Lemma 2), we know that any WC analysis performed on the composite SDF model is also valid for the refined TAI model (Propositions 1, 2 and 3).

Notice that in the first step of our method, we manually model the system as a TAI actor (or an actor composition). In this stage, no formal method is applied to verify that the TAI model is indeed a conservative model of the system. The development of the TAI model is meant to describe the behavior of the system based on our understanding, rather than on an automatic analysis of the code implementation. When possible, we directly model components using SDF. One could also try to always directly conservatively model the system as an SDF graph. However, the correspondence between the implementation and an SDF graph is often harder to observe, as internal events are often abstracted away from. The intermediate TAI modeling step increases the confidence we have in the reliability of the
model. For similar reasons, a TAI model could be an important step also in case the ultimate goal is to achieve automated analysis of a system, extracted from a system description (such as code, annotations, or description languages). Since TAI models are more general and expressive, it is more likely that TAI models could be more easily synthesized from system constructs.
Chapter 3

CompSOC

CompSOC is a predictable multiprocessor platform template. CompSOC achieves predictability by providing both (1) predictable resources, and (2) an SDF programming model for the user applications. Given an SDF user application and throughput constraints, a tool-flow is used to generate a platform hardware instance and a mapping of the application tasks to processors. In this chapter we provide a description of the platform and the existing models of the components. After clarifying the architecture, we present a model for a shared buffer under static-order scheduling. Then, we present a model for a task execution, and a top-level view on our communication-channel model.

3.1 CompSOC architecture

A CompSOC platform is composed of several basic building blocks, also called tiles. These communicate over a composable and predictable Network on Chip (NoC) [11]. A CompSOC platform typically includes processor tiles, memory tiles, a host tile and peripheral tiles. These are described further in [10]. All tiles are predictable and composable and can be modeled as Latency-Rate Servers [20]. We discuss these models further in Section 3.4, where we describe existing modeling of CompSOC components. The hardware platform is intended to deploy multiple applications, where each application consists of a set of tasks. The tasks may be mapped on multiple processors, to meet the performance constraints of the application. The CompSOC hardware micro-architecture is illustrated by an example instance of the platform appearing in Figure 3.1.

The instance in Figure 3.1 includes three processor tiles, two memory tiles, and two peripheral tiles. Each processor can run tasks from several applications. In each processor tile there is a single processor and several dedicated local memories: an Instruction Memory (IMEM), a Data Memory (DMEM), and input and output Communication Memories (CMEM-i and CMEM-o, respectively). Communication memories are used for inter-tile communication. The number of communication memories in the tile depends on the number of communication-channels defined for the specific instance, and on the choices made by the platform designer. Whenever communication is required between tiles, a communication-channel is required on the NoC. Based on the predefined channels and their bandwidth requirements, a NoC instance is generated. If a NoC instance exists, then it is configured according to the requirements, otherwise an instance is synthesized.

Whenever two tasks require communication, a dedicated FIFO needs to be instantiated. If both tasks reside on the same processor, then the communication can be performed using DMEM, in which case no communication-channel is required on the NoC. An exception is the case where the communicated data chunks are too large to fit in the local memory. In this case, the FIFO buffers are allocated in a shared memory external to the processor tile, requiring also a communication-channel to be allocated on the NoC. When the communicating tasks reside on different tiles, the FIFO can either reside in one of the tiles’ communication memories (typically, CMEM-i of the task reading the data, to reduce the time required to fetch data when performing a load instruction), or, when the data is too large to be kept on the local memory, an external shared memory can be used. In this case, two communication-channels will be defined on the NoC, one between each of the processor tiles and the memory tile. The protocol used for data-synchronization between the tasks is described in Section 3.2.

Prior to each execution of a task, any data to be processed by the task must reside locally inside the tile. A local memory access is performed via a dedicated interface, and always requires a fixed amount of time. In contrast, fetching data from a remote memory requires the services of the NoC and of the
Figure 3.1: CompSOC instance example. The instance consists of three processor tiles, two memory tiles, and two peripheral tiles.

(possibly shared) memory controller. These are designed to be composable while servicing multiple clients, and may result in a variable (yet predictable) response-time. As such, predicting their WC timing involves more complex modeling. We discuss their models further in Section 3.4. A processor tile is connected to the interconnect via a Network Interface (NI), which includes a buffer for outgoing NoC requests. The processor uses the aid of a Direct Memory Access (DMA) controller when performing external memory operations. The DMA has a queue for outgoing requests and is dedicated to a specific NI.

On top of the hardware (from the SW stack perspective), the CompSOC platform includes also the CompOSe [9] operating system. This thin operating system layer is entirely predictable. By using a TDM scheduler for the applications, the OS makes also the CPU composable. The operating system also includes an API for the intertask communication protocol. To comply with the modeling framework, an application must be programmed to comply with the SDF MoC, as explained in Section 3.6.

3.2 Intertask communication protocol

The software-level communication-channels between tasks are implemented using the C-HEAP communication protocol [16]. The protocol allows for communication of data between a producer-task and a consumer-task. The data is communicated in predefined quanta termed data tokens. The amount of data related to a token may vary in the system, but is fixed per C-HEAP channel. A data token produced by the producer is stored in a shared memory. The consumer then acquires the data and frees the space, allowing for production of more data. The communication between the producer and the consumer is separated into two different parts:

1. synchronization over the availability of data or space, and
2. the fetching and storing of data.

In the first part, the producer and the consumer signal the completion of data production and consumption by communication over the availability of full tokens and empty tokens. A full token, released by the producer and obtained by the consumer, indicates the availability of a data token in the FIFO. Similarly, an empty token, released by the consumer and obtained by the producer, indicates the availability of space in the FIFO. The producer and the consumer synchronize over full or empty tokens using the predefined C-HEAP synchronization primitives listed in Table 3.1.

Notice that obtaining a full token (using the claim_data) primitive includes only obtaining an indication that data is available in the FIFO, and does not include actually reading the data token.
Table 3.1: C-HEAP synchronization primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Initiator</th>
<th>Blocking/non-blocking</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>release_data</td>
<td>producer</td>
<td>non-blocking</td>
<td>signal the completion of data token production</td>
</tr>
<tr>
<td>claim_data</td>
<td>consumer</td>
<td>blocking</td>
<td>obtain a full token from the FIFO</td>
</tr>
<tr>
<td>release_space</td>
<td>consumer</td>
<td>non-blocking</td>
<td>signal the completion of data token consumption</td>
</tr>
<tr>
<td>claim_space</td>
<td>producer</td>
<td>blocking</td>
<td>obtain an empty token from the FIFO</td>
</tr>
</tbody>
</table>

The C-HEAP protocol layer, as illustrated in Figure 3.2, acts as a communication FIFO between a producer and a consumer. The term communication-channel in this context is not to be confused with a communication-channel on the NoC. Whereas here the channel is at the application-level, a NoC communication-channel depicts the connection between two tiles in the platform, which is at the lower hardware level. For example, it is possible that a consumer and a producer communicate using a remote memory located in a memory tile. In this case, a single C-HEAP communication-channel requires two NoC channels: one between the producer and the memory, and another between the consumer and the memory.

The implementation of the C-HEAP protocol requires a cyclic buffer for storing the data tokens, and two additional administrative memory locations: a write pointer writec and a read pointer readc. The pointer writec indicates the next location in the buffer that is available for the producer, and the pointer readc indicates the location in the buffer containing the next data token to be read by the consumer.

When the consumer requires a data token, it issues a call to the blocking claim_data (1), in order to obtain a full token. The consumer then blocks until a full token is available. After the producer finishes producing a token, it indicates so by issuing a call to the non-blocking release_data (2). When a full token is available for the consumer, the claim_data call returns a pointer to the next data token (3), denoted in Figure 3.2 by *data. After a consumer finishes consuming the data token, it issues a non-blocking call to release_space (4). Before a producer produces data, it must ensure that space is available. The producer does so by obtaining an empty token by issuing a call to the blocking claim_space primitive (5). When an empty token is available in the buffer, claim_space returns a pointer to the next available free location in the buffer (6), denoted in Figure 3.2 by *space. Let $F$ be the size of the buffer. Notice that since initially the buffer is empty, hence initially $F$ empty tokens are available. As such, the first $F$ calls by the producer to claim_space do not require prior releases of space by the consumer.

Listing 1 provides typical code segments for a consumer and a producer.
A consumer calling `claim_data` blocks until a full token is available. In practice, `claim_data` is implemented by continuously polling the values of `writec` and `readc` and calculating the difference between them. Whenever `writec > readc`, it implies that data is available for consumption. Similarly, a producer polls the values of `writec` and `readc` until space is available in the buffer. Given that the buffer has a known initial capacity of `F` data tokens, the difference between `writec` and `readc` indicates the space available: whenever `writec - readc < F`, space is available for the producer. A typical implementation is illustrated by Listing 2.

```c
int* claim_space(){
    while(writec = readc + F){}
    return writec;
}
```

```c
int* claim_data(){
    while(readc = writec){}
    return readc;
}
```

Listing 2: Typical implementations for the `claim_space` and `claim_data` primitives.

Releasing data or space is simply done by incrementing the relevant pointer. Notice that the producer alone updates `writec`, and symmetrically, the consumer alone updates `readc`. Typical implementations of `release_data` and `release_space` are provided in Listing 3. The pointers count in units of single data tokens.

```c
void release_data(){
    writec++;
}
```

```c
void release_space(){
    readc++;
}
```

Listing 3: Typical implementations for the `release_data` and `release_space` primitives.

### 3.2.1 Shared FIFO and counters memory allocation

The actual physical memory location of the FIFO and of the administrative counters is flexible and depends on the design choices made in the specific platform instance. Typically, duplicate copies of the administrative counters exists in the platform. As the producer side is responsible for updating the write counter `writec` by issuing a call to `release_data`, the write counter is kept locally at the output communication memory of the producer tile. A copy of the counter typically exists also at the input communication memory of the consumer side. When `release_data` is called, the local copy in the producer tile is updated first, and then the remote copy at the consumer tile is updated with the aid of a DMA. When issuing a call to `claim_data`, the consumers polls locally on the duplicated counter. Symmetrically, in a typical implementation, the read counter is only updated by the consumer thus a local copy of `readc` lies in the consumer tile and a duplicate in the producer tile.

It is important that the value of `writec` used in the primitive `claim_data` reliably represents the status of the FIFO. Namely, `writec` should only be updated to signal the availability of data after the data has already been written to the FIFO. If `writec` is updated too early, `claim_data` may prematurely return, and the consumer may read invalid data. Recall that on the producer side, `release_data` is only called by the producer after writing the data to memory. However, since write requests are non-blocking, the producer does not wait for an acknowledgement for the completion of writing the data token before calling `release_data`. In theory, this could bring rise to a potential race condition, but since the interconnect is guaranteed to perform transactions in order, as long as the data FIFO and the copy of `writec` used in `claim_data` are located in the same memory, memory consistency is guaranteed.
In the default setting, the shared FIFO is located on the consumer tile, however this configuration is not restrictive. Specifically, when the size of the FIFO is too large to fit in local memory, the FIFO is allocated in a shared memory, located in a memory tile. If a copy of writec is kept at the consumer tile, the abovementioned consistency issue arises. One possible solution is to verify that the data token is written before writing the copy of writec. This can be achieved by issuing a read request to the FIFO after writing the data token. Since transactions are performed in-order, the arrival of a completion for the read request indicates that the write request has also been completed. To refrain from this additional read request, it is also possible to keep a copy of writec in the shared memory instead of the copy in the processor tile. Another option is to refrain from keeping a copy and polling directly on the producer’s copy. The various flavors of allocation the FIFO and the administrative counters require different models, which are provided in Section 5.3.

3.3 Programming model and platform synthesis

A CompSOC instance is generated using a tool-flow mainly based on the flow suggested in [22]. In this section, we overview the tool-flow, focusing on parts relevant to our work, and explain where our models are integrated.

A user is required to use SDF as a programming model when designing an application. As such, a user application is designed as a task-graph where the two following requirements are imposed on each of the tasks in an application:

1. All resources required by a task are claimed before execution, and all used resources and produced data are released after execution. Any data required during execution must exist locally at the processor tile in which the task executes before the task begins its execution.

2. A task must produce and consume a fixed amount of data in each execution.

Figure 3.3 provides an overview of the tool-flow. A user is required to provide the following inputs (1-3 in Figure 3.3):

1. A specification of the available Hardware resources. This includes the number of processors, the memory layout, and the existing NoC communication-channels and their available bandwidth.

2. An SDF model of the application, and C code for each of the actors in the graph.

3. Throughput requirements.

Given the above inputs, an execution of the tool-flow results in a feasible platform instance, i.e., a platform that is guaranteed to keep the provided throughput constraints using only the available hardware resources. The generated platform includes the hardware (VHDL code that is synthesized on an FPGA platform), a mapping of tasks to tiles, and a schedule for the tasks.

Recall that in an SDF graph edges are assumed to have infinite buffer capacities. However, in real-life situations communication buffers are bounded, and memory must be allocated to contain any buffered data. Given an SDF graph, it is possible to calculate for each edge the required buffer size [19]. If less memory is allocated for the buffer, a task (an actor) needs to block more frequently when the buffer is full, resulting in throughput degradation or ultimately a deadlock. As memory is a limited resource in an actual platform, it is desirable to minimize the buffer sizes. This creates a tradeoff between memory and throughput [21].

The tool iteratively attempts to bind the tasks to processors. First, a crude analysis is performed on the application SDF graph. The analysis assumes no communication overhead, i.e., the channels are assumed to have infinite bandwidth and no latency. This stage is indicated in Figure 3.3 as memory/throughput tradeoff (4). This analysis results in a set of memory-throughput tuples, each keeping the given throughput constraints (5). Each tuple consists of a memory distribution (i.e., buffer allocations) and a corresponding WC throughput. The tool iteratively selects one of the memory distributions and maps the tasks to tiles such that the hardware resources (e.g., memory sizes) are also considered. This binding of tasks to tiles is indicated in Figure 3.3 as the mapping stage (6). In each iteration of the flow, a different mapping of tasks to tiles is chosen (7).

The application SDF graph supplied by the user includes only the execution-times of the tasks on the CPU. However, additional overhead, such as task scheduling and communication must also be considered. As such, the next stage in the flow is to create an SDF graph that also includes communication
Figure 3.3: CompSOC Tool-flow
overheads (reading and writing data, and data/space synchronization). To this end, the tool uses the specification of the hardware resources to calculate and assign values to a parametrized channel-model. The tool generates an SDF graph based on the original application graph, where all channels are substituted by SDF channel-models, as illustrated by Figure 3.4.

![Application graph: channel-model is abstracted](a)

![Binding-aware: channel-model is refined](b)

Figure 3.4: The tool-flow refines the channels (edges) of the application SDF graph into a binding-aware channel.

The tool uses existing parametrized models of platform components, denoted by platform models (8), to create a channel model (9). The channel model is composed with the application tasks to create a low-level SDF graph (10). The models contributed in this thesis are to be integrated with this part of the tool-flow. As the SDF graph of the system now takes into account the communication and scheduling overheads caused by the binding of tasks to tiles, we denote it as a binding-aware model (11).

If the throughput constraints hold for the binding-aware SDF model, then the flow terminates successfully (12). If the throughput constraints no longer hold after considering the communication overhead (13), the tool selects a different memory distribution from the memory-throughput set and performs another iteration. A successful termination results in a platform instance: a mapping of the tasks to tiles, we denote it as a binding-aware model (11).

In the rest of this chapter we explain how to model the various platform components, to enable the construction of the binding-aware SDF model. First, we overview the existing models for the CPU, the interconnect and the memory. Then we explain how to compose these to model memory operations, which are used for inter-task communication. Finally, we explain how to use such compositions in order to flatten an SDF application graph into a binding-aware SDF graph.

### 3.4 Conservative modeling of hardware resources

As already mentioned, the predictability of CompSOC is achieved by formally modeling the system components and their composition. Each of the components is modeled as an SDF graph (or an SDF extension, e.g., CSDF [4]) and the components are composed to a closed SDF graph on which performance analysis is performed (the binding-aware graph). As the components are conservatively modeled as time-deterministic SDF graphs, the composite SDF graph is monotonic [26] and hence performance analysis performed on the model provides WC performance guarantees. In this section we provide conservative models for the CPU, the interconnect and the shared memory. These components are used in the low-level model of the binding-aware SDF graph.

#### 3.4.1 Processor

The CPU is shared among applications, using TDM scheduling imposed by the operating system. A TDM scheduler is a Latency-Rate server (LR-server) [20], which can be conservatively modeled as a dataflow graph [25]. Figure 3.5 provides a simple SDF model for an LR-server.
Given the TDM wheel and the number of slots allocated for an application, the computation-time of a sequence of instructions performed on the CPU can be translated to a Worst-Case Response-Time (WCRT). As such, the execution of a sequence of instructions on the CPU can be modeled as single SDF actors with an execution-time equal to the WCRT on the CPU, as depicted in Figure 3.6.

When modeling the execution of a task on the shared CPU, the execution-time of a task is used to extract the response-time of the task. Whenever the sequence of operations includes activities that require the service of shared resources outside the processor tile, the execution needs to be divided such that each phase in the execution encapsulates internal operations only. We discuss this point further in Section 3.6.

3.4.2 Interconnect

In [11] several SDF models are presented for the NoC, with various levels of detail. The NoC provides TDM scheduling to its clients. In addition to the time-slices dedicated to the application, the overhead of headers, and the dependency on flow-control are also considered in the models. Although more detailed models exist, for our models we use a simple LR-server to model the NoC propagation. We do so in order to keep our models brief and readable. The Models are modular and the NOC model can be replaced for a more refined one if needed.

We use a parametrized NoC model. The exact values for the latency and rate of a NoC channel can be extracted based on the NoC instance. We uniquely identify the NoC channel and the traffic direction by the end-points of the channel: $I$ and $T$, denoting the initiator and target of transactions. Traffic traveling from the initiator to the target is denoted by the sequence $IT$, and traffic in the opposite direction is denoted by the sequence $TI$.

A NI buffer with limited capacity is located at the NoC input. To model buffer back-pressure, we augment the latency-rate actors with a dashed edge from the rate-actor to the output $c_{out}$. To model incoming back-pressure from other components, we add the input $c_{in}$.

A NI buffer with limited capacity is located at the NoC input. To model buffer back-pressure, we augment the latency-rate actors with a dashed edge from the rate-actor to the output $c_{out}$. To model incoming back-pressure from other components, we add the input $c_{in}$. 

---

**Figure 3.5**: General dataflow model for an LR-server

**Figure 3.6**: Modeling the execution of a sequence of instructions on the virtualized CPU

**Figure 3.7**: NoC model for a traffic traveling from source node $K$ to target node $L$. 

---

26
The delays of the actors $N_{θKL}$ and $N_{ρKL}$ depend on the slot allocations for this channel, as described in further detail in [11].

### 3.4.3 Memory controller

The memory tiles contain memory controllers utilizing composable arbitration. For example, a TDM scheduler, or Credit-Controlled Static Priority (CCSP) with delay blocks [1]. Data arriving at the memory interface is written to memory in data chunks called *atoms* or Memory Service Unit (MSU)s. The allocated bandwidth and the initial latency can be conservatively modeled as a $LR$-server. These can be further refined and be modeled as a bi-rate server in [18]. For brevity we use a simple $LR$-server for our models. Figure 3.8 depicts a $LR$-server for the memory. The delays of the rate and the latency depend on the service guarantees of client $C$.

![Figure 3.8: Latency-rate model for the memory.](image)

The model of Figure 3.8 conservatively models the guaranteed service time a request encounters when accessing the memory. It is important to note that for composability reasons (which are beyond the scope of this document), if the response-time of the memory is better (shorter) than the response-time guaranteed by the latency-rate model then *delay blocks* are added. These delay the response such that the over-all response time is equal to that of the model.

### 3.4.4 Model compositions

The individual SDF models of the system components are composed together to model various system behaviors. In this section we provide an example of a composition that models the behavior of a write transaction. This specific example is essential, since the writing procedure is in fact often a part of many other system procedures. As such, we also use similar models when modeling C-HEAP primitives in Section 5. We summarize the units used throughout our models in Table 3.2.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>data token size</td>
</tr>
<tr>
<td>DMABS</td>
<td>DMA burst size</td>
</tr>
<tr>
<td>flit</td>
<td>NoC service unit</td>
</tr>
<tr>
<td>MSU</td>
<td>memory service unit</td>
</tr>
</tbody>
</table>

Table 3.2: Units summary

Figure 3.9 models a write access performed by a task executing on a processor $A$. After the task finishes its execution it writes the generated data to a remote memory. The parameters used in the model are summarized in Table 3.3.

Internally in the processor tile, the write transaction requires the interaction of several components. First, in order to execute a data-producing task, the task must be able to write the produced data to a local buffer, located on CMEM-o. Then, a DMA request is issued to post the data on the network-interface queue. The abovementioned components are modeled as the actors $A$, buf, and DMA. The actors and parameters in the model are used to either model the execution-time of system components (e.g., the execution on the CPU or the delay of a DMA execution), to model predefined buffer sizes (e.g., the local buffer, the number of pending DMA requests, and the NI buffer), and also to model unit-conversion.
Table 3.3: Write transaction model parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer/queue sizes</td>
<td>$B_A$</td>
<td>size of local buffer at the processor tile of the source actor</td>
</tr>
<tr>
<td></td>
<td>$B_{NI}$</td>
<td>network-interface buffer size</td>
</tr>
<tr>
<td></td>
<td>$B_M$</td>
<td>Memory buffer size</td>
</tr>
<tr>
<td></td>
<td>$R$</td>
<td>maximal number of pending DMA requests</td>
</tr>
<tr>
<td>Rates</td>
<td>$r$</td>
<td>number of tokens produced by the source actor in a single firing</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>number of DMA burst required for a single data token</td>
</tr>
<tr>
<td></td>
<td>$d$</td>
<td>number of flits required for a single DMA burst</td>
</tr>
<tr>
<td></td>
<td>$m$</td>
<td>number of flits in a single memory access</td>
</tr>
</tbody>
</table>

between different components (e.g., from data token-size to DMA burst). If actors are only used to model buffer capacities or unit-conversions, then they have an execution-time of 0.

![Diagram](image)

Figure 3.9: Writing from A to memory

In each firing, the task produces $r$ data tokens. The edge (buf, $A$) models the size of the local buffer. The edge has an initial number of $B_A$ tokens, representing the number of data tokens the local buffer can contain. Each produced data token is transformed into $b$ DMA bursts, as indicated by the rates on the edge (buf, DMA). The DMA has a request-queue of size $R$, which is modeled by the edge (DMA, buf) with an initial number of $R$ tokens. The actor buf is only intended to model the unit-conversion, hence it has an execution-time of 0 time units. The actor DMA models the delay required for the DMA to transmit a single burst. The DMA executes sequentially, hence the self-edge. Each burst of the DMA is transformed into $d$ flits. The size of the NI buffer is modeled by the parameter $B_{NI}$. Each firing of the NoC rate-actor $N_{\rho_{A,M}}$ indicates that a data unit of size flit has experienced been serviced, and thus surely has left the NI buffer.

The NoC model (the actors $N_{\theta_{A,M}}$ and $N_{\rho_{A,M}}$) is connected to a memory model (the actors $M_{\theta_{A}}$ and $M_{\rho_{A}}$), which are latency-rate abstractions as explained in Sections 3.4.2 and 3.4.3 respectively. The actor collect, located at the output of the memory model, is intended to model the transformation of memory-service units to data-token units. The original data-token is split into smaller data-chunks several times during the write transaction, and we are typically interested in modeling the writing of an entire token. As such, the actor collect only fires after $c$ tokens have been serviced by the memory, where $c$ is the number of memory service units required for a single token.

Models similar to the one provided in Figure 3.9 are used in the existing channel-model of the CompSOC tool-flow. The model assumes no interference between tasks of the same application. However, in the platform a single DMA is shared among all tasks of the same application (executing on the same tile). As such, the DMA is in fact a shared-resource with a shared request-queue. So far the reasoning behind excluding task interference on the DMA has been the assumption that the DMA-request-queue can be modeled as an infinite buffer. In the next section we show how the interference on a shared-buffer (or queue) can be modeled, provided that the execution of the tasks sharing the buffer is statically-ordered.
are about to be fired by
t
In the example, the actors
d
However, the DMA is shared among all tasks of the
same
service). For composability of the applications, each application in the system has a dedicated DMA.
(in constrast to a composable arbitration scheme, where each client would be guaranteed to have certain service). For composability of the applications, each application in the system has a dedicated DMA.
However, the DMA is shared among all tasks of the same application. CompSOC is not only a composable platform, but also a predictable one. To achieve predictability for an application, we must consider any possible interference between tasks sharing resources. Sharing the DMA implies both temporal and spatial interference between tasks. We explain how to model these on a shared buffer. The spatial interference in the case of the DMA HW block is in the DMA request-queue, which is slightly different than a buffer, as we discuss in the last part of this section. First, we explain how to model a shared buffer. Then, we generalize the model to fit also the case of a request-queue.

In this section we explain how to model a buffer shared over multiple clients, provided their static-order schedule. To illustrate this, we use an example with three task actors, $t_0$, $t_1$, and $t_2$, ordered according to their indices, as depicted in Figure 3.10(a).

Temporal interference
The buffer services its clients sequentially, i.e., a single request is issued by the buffer at a time. To model this temporal interference, we require a single buffer-actor per task, as illustrated by Figure 3.10(b). In the example, the actors $d_0$, $d_1$, and $d_2$ are added, where each firing of task $t_i$ releases $r_i$ tokens to the corresponding buffer-actor. Each buffer-actor models the processing of a single buffer request, and as such they all have the same execution-time. Since the buffer handles requests in a FIFO manner, buffer-actor firings are ordered. This is modeled by the edges between the buffer-actors. A buffer-actor $d_{i+1}$ only fires after actor $d_i$ has fired $r_i$ tokens. To model this we place the actors $c_i$, that collect $r_i$ tokens from the corresponding buffer-actor $d_i$. A firing of an actor $c_i$ enables the execution of $r_{i+1}$ firings of the consecutive buffer-actor $d_{i+1}$.

Spatial interference
To model the spatial interference we take advantage of the fact the tasks are statically ordered. Since in each execution a fixed number of tokens is fired, equivalent to a certain number of buffer requests, and since the buffer serves the requests in a FIFO manner, we know the exact sequence of requests generated by the buffer. To illustrate this using our example, we color the requests produced by the buffer according to their origin. Requests originating from $t_0$ are colored in red, from $t_1$ in blue, and from $t_2$ in orange, as illustrated by Figure 3.10(c). The sequence of buffer requests is known in advance, as is also illustrated by Figure 3.10(c).

To conservatively model the spatial interference, we ensure that each task-actor in the model begins its execution only after sufficient room is available in the buffer. To allow a firing of a task-actor, we simply count the number of buffer requests that must have been serviced before the firing. Since the firings are statically-ordered, this information is available at design-time.

Let $B$ be the size of the shared buffer. In order to have enough room for the firing of task-actor $t_i$, at least $r_i$ slots in the buffer must be available. Since the requests are processed in a FIFO manner, these are the last, or bottommost, slots in the buffer. In Figure 3.10(c) we illustrate the minimal requirement on the status of the buffer to allow the $n$-th firing of task $t_i$. In the example, $i = 0$ and $n = 3$, i.e., the task-actor is $t_0$, colored in red and requiring two slots in the buffer, and this is the task’s third firing. We know in advance the order of requests serviced by the buffer. We mark the number of requests that have entered the buffer before the firing the $n$-th firing of $t_i$ as $T_{n,i}$. The buffer can hold up to $B$ requests in total, and at the time of firing it must hold at most $B - r_i$ requests, to be able to contain $r_i$ requests that are about to be fired by $t_i$. The minimal number of requests that must have been serviced at the time of the firing is marked in Figure 3.10(c) as $S_{n,i}$, where

$$S_{n,i} = T_{n,i} - (B - r_i)$$
We denote the minimal number of requests of type \( j \) that must have been serviced by the buffer before the \( n \)-th firing of actor \( t_i \) as \( s_{n,i,j} \), which is the number of \( j \) bursts in \( S_{n,i} \). This number can be expressed as follows:

\[
s_{n,i,j} = \begin{cases} 
    n \cdot r_j - I_{i,j} & \text{if } j < i \\
    (n - 1) \cdot r_j - I_{i,j} = n \cdot r_j - r_j - I_{i,j} & \text{if } j \geq i
\end{cases}
\]

where \( I_{i,j} \) is the maximal number of requests that exist inside the buffer at the moment of firing, i.e.,
it is the number of requests of type \( j \) in the window of \( B - r_i \) requests before \( T_{n,i} \). We make two observations. First, \( I_{i,j} \) is independent of \( n \) which can be calculated at design-time. Second, there is a linear dependency between the number of enabled \( t_i \) executions and the number of tokens generated by \( d_j \), with a rate of one execution of \( t_i \) for every \( r_j \) executions of \( d_j \). We use these observations and place dependency edges originating from each buffer-actor \( d_j \) to each task-actor \( t_i \), with an outgoing rate \( 1 \) and an incoming rate \( r_j \). The additional constant in the equation for \( s_{n,i,j} \) (\( -I_{i,j} \) for the case where \( j < i \) and \( -r_j - I_{i,j} \) otherwise) indicates that some executions of \( t_i \) are already enabled before any token generation on the output of \( d_j \). We model these as initial tokens on the edge from \( d_j \) to \( t_i \).

To calculate \( I_{i,j} \) we count the number of type \( j \) requests in a window of size \( B - r_i \) preceding \( T_{n,i} \), i.e., the window between \( S_{n,i} \) and \( T_{n,i} \). To have simpler variable indices in the summations below, we separate between the case where \( j < i \) and the case where \( j \geq i \). Let \( N \) be the number of tasks in the schedule, and let \( R \) be the sum of all requests in a full cycle, i.e., \( R = \sum_{k=1}^{N} r_k \). \( I_{i,j} \) is the following somewhat tedious expression:

\[
I_{i,j} = \left[ \frac{B - r_i}{R} \right] \cdot r_j + \min(\max((B - r_i) - \frac{B - r_i}{R} \cdot R - \sum_{k=1}^{(N+i-j-1)\%N} r_{(j+k)\%N}, 0), r_j) \tag{3.1}
\]

The expression \( \left[ \frac{B - r_i}{R} \right] \) is the number of full cycles of firing (of size \( R \)) that can be contained in \( B - r_i \) slots in the queue. The expression \( \left( \frac{B - r_i}{R} \right) \cdot r_j \) simply denotes that for each whole cycle we have \( r_j \) tokens of type \( j \). For the remainder, \( (B - r_i) - \left( \frac{B - r_i}{R} \right) \cdot R \), we need to count the number of tokens of type \( j \). To that end, we subtract the sum of all tokens in the buffer remainder that appear after the tokens of type \( j \). A negative result implies no tokens of type \( j \) exist in the remainder, hence we take the maximum between the result and \( 0 \). A result greater than \( r_j \) indicates that tokens of tasks executed before task \( t_j \) also exist in the remainder. At most \( r_j \) tokens of type \( j \) can exist in the remainder, hence we take the minimum of the previous result and \( r_j \).

The final result is a model where a dependency edge \((d_j, t_i)\) exists for all \( i, j \leq N \). The outgoing rate of the edge is \( 1 \), the incoming rate is \( r_i \) and the number of initial tokens is:

\[
i_{i,j} = \left\{ \begin{array}{ll}
i_{i,j} & \text{if } j < i \\
r_j + I_{i,j} & \text{if } j \geq i
\end{array} \right.
\]

It is easy to check that this edge enables the \( n \)-th firing of \( t_i \) exactly after the \( s_{n,i,j} \)-th firing of \( d_j \). For our example, the final model appears in Figure 3.10(d).

So far we have explained how to model a shared buffer. In CompSOC, the spatial contention on the DMA is in the form of a shared request-queue, and not a buffer. The DMA has a request-queue, shared among the tasks. Each firing of a task is translated to a (known) number of DMA requests. The transition from a shared buffer model to a shared request-queue model is simple, as illustrated in Figure 3.11. The dependency edge from \( d_j \) to \( t_i \) is augmented with an additional actor \( h_{j,i} \). The actor is responsible for collecting a number of DMA-actor firings that is equivalent to the number of firings required for a single slot in the request-queue. For each \( d_j \) actor the number of firings required to free a single slot in the queue is \( r_j \). Only after \( r_j \) firings took place, the collecting actor \( h_{j,i} \) enables the release of a token on the edge between \( h_{j,i} \) and \( t_i \).

The number of initial tokens can be derived by an equation similar to that provided for \( i_{i,j} \), following similar reasoning. We simply replace the following parameters in equation (3.1):

- \( r_i \to 1, r_j \to 1 \). The number of slots in the request-queue is always one per execution of a task.

![Figure 3.11: Back-pressure edge from a shared buffer vs. back edge from a shared request-queue](image-url)
• $R \rightarrow N$. The size of a request-cycle is now $N$, as every request requires a single slot.

The maximal number of slots occupied in the request-queue by requests of type $j$ at the time of a firing of task $i$ is thus as follows:

$$I'_{i,j} = \left\lfloor \frac{B - 1}{N} \right\rfloor + \min(\max((B - 1) - \left\lfloor \frac{B - 1}{N} \right\rfloor \cdot N - (N+i-j-1)\%N, 1, 0), 1)$$ (3.2)

From this it follows that the number of initial tokens on an edge $(h_{j,i}, t_i)$ is $i'_{i,j}$, where:

$$i'_{i,j} = \begin{cases} 
I'_{i,j} & \text{if } j < i \\
1 + I'_{i,j} & \text{if } j \geq i
\end{cases}$$

We have shown how to model both a shared buffer and a shared request-queue. The shared request-queue model is essential for modeling the interference of statically-ordered tasks on the DMA. The shared buffer model could also be used in CompSOC for modeling interference on the buffer of an NI port. Currently, it is not allowed for different tasks to share NI ports. However, if this requirement is relaxed in the future, our models enable analysis of such interference.

### 3.6 SDF execution model

To allow for formal analysis, an application programmer must ensure that an application complies with the SDF MoC. To that end, it must be possible to model the application as an SDF graph, where each of the actors in the graph is a task in the application. Each task must be implemented to behave as an SDF actor. Namely, a task must first obtain all required data and space before beginning its execution. In addition, only after execution has finished, the task releases any data generated, and any space that has been made available. The sequence of operations in the task is then the sequence depicted in Figure 3.12 (identical to Figure 1.1).

![Figure 3.12: The sequence of operations in the SDF programming model.](image)

The execution phase of a task includes only operations that are performed locally on the processor tile on which the task is scheduled. Since all operations are performed locally (and since all local memories are dedicated to the processor), it is possible to extract the number of clock cycles required for execution of the task on the processor. Given the execution-time and the TDM schedule, a WCRT can be calculated for the task.

The application-level model contains only the execution phase of the tasks. The platform resources used for communication are abstracted, and communication-channels between tasks (expressed as edges in the SDF application graph) are assumed to have infinite capacity. However, when an application is deployed on the platform, physical communication-channels with finite buffers are used. To comply with the physical communication-channels, an application is required to use the C-HEAP synchronization primitives for intertask communication (see Section 3.2). Before beginning its execution, a task claims an empty tokens on each of its output channels (buffers) and claims a full tokens on each of its input channels. Claiming a full token on an input channel is always followed by reading the actual data from the buffer.

When a task has multiple incoming and outgoing edges, it performs claiming of tokens from all channels sequentially, as illustrated by Figure 3.13. For a task $t$ with $n$ outgoing edges and $m$ incoming edges (Figure 3.13(a)), the task first claims empty tokens from all outgoing edges (channels $c_1$ to $c_n$) and full tokens from all incoming edges (channels $p_1$ to $p_m$). Only after obtaining all required data and space, the task executes. Similarly, only after the task finished executing, it releases all data and space sequentially. In Figure 3.13(b), the task first claims all empty tokens and then claims all full tokens and reads the data. This behavior is not restrictive, as it is also allowed to claim the tokens in any other order, as long as all space and data is obtained before beginning execution (similarly for releasing and writing, which may only occur after the execution phase).
3.6.1 Software-level interaction with the communication-channel

As explained in Section 3.4.1, whenever we have a sequence of CPU instructions with a computation-time that depends solely on dependencies internal to the CPU, we can model this sequence as a single self-edged SDF actor with a delay equal to the WCRT. In the case of an SDF task (see Section 3.6), the execution phase is indeed suitable for such modeling, but the other phases may involve some external interaction.

Each of the five phases of an SDF task requires CPU activity, which we model as a single SDF actor, as illustrated by the chain of SDF actors (drawn as circles) in Figure 3.14. In addition, we add the additional actors initialize and terminate to model any scheduling overhead. In addition to the CPU, other dependencies also exist. Whenever a load instruction is performed from a remote memory, the model should be extended to include also the external dependencies on the interconnect, or on remote memories. An example for this is the read operation. If the FIFO is located on a remote memory, then the read operation includes such external dependencies. We model this using the cloud-block Read. Clouds in Figure 3.14 are meant to be place-holders for SDF models, which will be presented in Section 5. Ingoing and outgoing edges between SDF actors and clouds denote SDF channel-placeholders, but are not restrictive. It is possible that in the full model additional edges are present, and also that some edges are absent (in Section 5 we provide an SDF model for the place-holders and also for provide the precise interaction between them). In the case of the read activity, if the FIFO is located within the CPU tile, the model can be degenerated to a single SDF actor with an execution-time of 0.

Another case in which the response-time cannot be directly modeled by simply using an activity’s sequence of instructions (i.e., its code) is when the activity includes branching instructions and data dependencies. In this case the computation-time of a task may depend on the context (i.e., the entire system). An example for such behavior is the C-HEAP claiming procedures. The procedure could operate entirely locally within the CPU tile (e.g., when the polled counter is located locally), and still the number of polling iterations performed by the procedure is unknown without the context of a system. In addition, the polling procedure is non-monotonic, hence requires special treatment, as we thoroughly discuss in Section 4. In Figure 3.14, the C-HEAP claiming primitives are again modeled as cloud-blocks. In Chapter 5, where we provide a full channel-model, we provide their SDF models.

For all four phases other than execution phases, i.e., claim, read, write, and release, could potentially interact with components outside the processor tile, depending on the memory locations of the FIFO and the administrative counter. For this reason we add cloud-blocks and dependency edges for all of them. Figure 3.14 does not explicitly show interaction between the block-models, although dependencies between them could exist, as we explain further in Section 5.
Figure 3.14: SDF model for the execution phases combined with channel interaction. The channel model components appear as informal cloud-blocks.

A system has multiple C-HEAP communication-channels. To provide each communication-channel with a unique ID, we denote the C-HEAP channel between task $t_1$ and task $t_2$ as $(t_1, t_2)$, as illustrated by Figure 3.15.

Figure 3.15: The communication-channel between two tasks $t_1$ and $t_2$, denoted by $(t_1, t_2)$.

A task may have multiple incoming and outgoing edges, i.e., it may participate in multiple C-HEAP communication-channels. We summarize the input and output interfaces of the execution model of a task $t$ in Table 3.5. The interfaces are the call or the return from any of the cloud-blocks of Figure 3.14. In this case, the term return means that the upon only after an event on this port, the CPU is free to continue with other activities. Notice that this does not necessarily mean that the actual procedure has ended (e.g., in the case of a non-blocking procedure, the procedure may continue in parallel to the
We define a set of input and output ports for each communication-channel in which a task \( t \) participates. A different set of ports is defined for the case where a task acts as a producer or as a consumer.

### Table 3.4: Input and output ports for CPU model of task \( t \)

<table>
<thead>
<tr>
<th>Role/Channel</th>
<th>Port</th>
<th>Direction</th>
<th>Call/return</th>
<th>System event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer / ( (p,t) )</td>
<td>( t_{CD(p),c} )</td>
<td>out</td>
<td>call</td>
<td>claim_data</td>
</tr>
<tr>
<td></td>
<td>( t_{CD(p),r} )</td>
<td>in</td>
<td>return</td>
<td>claim_data</td>
</tr>
<tr>
<td></td>
<td>( t_{Read(p),c} )</td>
<td>out</td>
<td>call</td>
<td>read_data</td>
</tr>
<tr>
<td></td>
<td>( t_{Read(p),r} )</td>
<td>in</td>
<td>return</td>
<td>read_data</td>
</tr>
<tr>
<td></td>
<td>( t_{RS(p),c} )</td>
<td>out</td>
<td>call</td>
<td>release_space</td>
</tr>
<tr>
<td></td>
<td>( t_{RS(p),r} )</td>
<td>in</td>
<td>return</td>
<td>release_space</td>
</tr>
<tr>
<td>Producer / ( (t,c) )</td>
<td>( t_{CS(c),c} )</td>
<td>out</td>
<td>call</td>
<td>claim_space</td>
</tr>
<tr>
<td></td>
<td>( t_{CS(c),r} )</td>
<td>in</td>
<td>return</td>
<td>claim_space</td>
</tr>
<tr>
<td></td>
<td>( t_{Write(c),c} )</td>
<td>out</td>
<td>call</td>
<td>write_data</td>
</tr>
<tr>
<td></td>
<td>( t_{Write(c),r} )</td>
<td>in</td>
<td>return</td>
<td>write_data</td>
</tr>
<tr>
<td></td>
<td>( t_{RD(c),c} )</td>
<td>out</td>
<td>call</td>
<td>release_data</td>
</tr>
<tr>
<td></td>
<td>( t_{RD(c),r} )</td>
<td>in</td>
<td>return</td>
<td>release_data</td>
</tr>
</tbody>
</table>

#### 3.6.2 Channel top-level view

Complementary to the task’s interface, we provide a high-level view of an inter-task communication-channel at the software-level, depicted in Figure 3.16. The model is composed of the following components: CS, CD, RS, RD, Write and Read. Notice that this is a software-level model since it abstracts from the HW-overheads. It does not model any specific HW components, or the interaction between them. Instead it models the sequence of high-level operations performed in the channel. We summarize the ports of a communication-channel \((t_1, t_2)\) in Table 3.5. The table also includes internal ports for modeling events between different modules in the model (e.g., between write and release). Since the model is at the software-level, the ports are not restricting, and it is additional that more ports and dependency edges exist in the final models of the primitives, which are provided in Section 5.

![Figure 3.16: Channel interface](image)

We conclude this chapter with an illustration of the execution of a task with multiple incoming and outgoing communication-channels. If a task has multiple incoming and outgoing edges, the CPU activities interacting with the communication-channels are performed sequentially, as explained in Section 3.6. Figure 3.17 demonstrates a model for a task \( t \) with \( m \) incoming edges and \( n \) outgoing edges (task \( t \) of Figure 3.13(a)).
Table 3.5: Input, output and internal ports for a channel \((t_1, t_2)\)

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>System event</th>
</tr>
</thead>
<tbody>
<tr>
<td>((t_1, t_2)_{RS,c})</td>
<td>in</td>
<td>call to release_space</td>
</tr>
<tr>
<td>((t_1, t_2)_{RS,r})</td>
<td>out</td>
<td>call to release_space</td>
</tr>
<tr>
<td>((t_1, t_2)_s)</td>
<td>internal</td>
<td>read updated by release_space</td>
</tr>
<tr>
<td>((t_1, t_2)_{CS,c})</td>
<td>in</td>
<td>call to claim_space</td>
</tr>
<tr>
<td>((t_1, t_2)_{CS,r})</td>
<td>out</td>
<td>return from claim_space</td>
</tr>
<tr>
<td>((t_1, t_2)_{RD,c})</td>
<td>in</td>
<td>call to release_data</td>
</tr>
<tr>
<td>((t_1, t_2)_{RD,r})</td>
<td>out</td>
<td>call to release_data</td>
</tr>
<tr>
<td>((t_1, t_2)_d)</td>
<td>internal</td>
<td>write updated by release_data</td>
</tr>
<tr>
<td>((t_1, t_2)_{CD,c})</td>
<td>in</td>
<td>call to claim_data</td>
</tr>
<tr>
<td>((t_1, t_2)_{CD,r})</td>
<td>out</td>
<td>return from claim_data</td>
</tr>
<tr>
<td>((t_1, t_2)_{Write,c})</td>
<td>in</td>
<td>start writing data</td>
</tr>
<tr>
<td>((t_1, t_2)_{Write,r})</td>
<td>out</td>
<td>finished writing data</td>
</tr>
<tr>
<td>((t_1, t_2)_{Read,c})</td>
<td>in</td>
<td>start reading data</td>
</tr>
<tr>
<td>((t_1, t_2)_{Read,r})</td>
<td>out</td>
<td>finished reading data</td>
</tr>
</tbody>
</table>
Figure 3.17: Execution illustration of a task with multiple channels.
Chapter 4

Non-monotonic abstraction

In the previous sections we established the background required for understanding the system behavior and the modeling techniques used in the CompSOC platform. In the platform, non-monotonic behavior exists in the polling procedure of the C-HEAP primitives `claim_data` and `claim_space`. In this chapter we confront the challenge of conservatively modeling non-monotonic behavior using the formalisms described in Section 2. We begin by providing a model using the theory of timed actor interfaces. We then show that the behavior is indeed non-monotonic and provide a conservative SDF abstraction.

4.1 Data abstraction

Our objective is to conservatively model the implementation of the C-HEAP protocol as an SDF graph. To that end, we use an intermediate TAI model of C-HEAP. The low-level TAI actors abstract away from data values and handle only sequences of events on their inputs and outputs.

Recall that the C-HEAP protocol is concerned with passing tokens of data using a shared buffer. Each token represents an amount of data, and the protocol is not affected by the values of the data itself, i.e., it abstracts away from the data shared over the buffer. On the other hand, in a C-HEAP implementation the values of `writec` and `readc` are used, making the implementation data dependent. In the synchronization primitives, the values of `readc` and `writec` are compared and their values indicate the availability of tokens. As TAI models are only expressive enough to represent sequences of events, in our model we abstract away from the data dependency existing in the Implementation.

This abstraction is possible since although the decisions in the implementation are data-dependent and rely on the values of the pointers, these values in fact represent a state of the system that strictly depends on previous synchronization events, rather than on actual data values. The values of the pointers represent the amount of data-tokens available in the buffer. This amount simply depends on the number of production and consumption events that occurred in the past. In our models, we abstract away from the values of the pointers and only model the meaning behind them.

Although we abstract away from the values of `writec` and `readc`, we still model the delays of the memory accesses made to retrieve them. Our model is a timed-model and is used to contract performance-bounds. The memory accesses to the pointers affect the timing of the system and thus must be taken into account, regardless of whether or not the actual values are used in the model.

4.2 `claim_data`: non-monotonic abstraction

In this section we present a model for the C-HEAP primitive `claim_data`. We require a new basic actor, the conditional branch actor.
4.2.1 The conditional branch actor $CB_{n>0}$

In this section we present an actor for conditional branching that could be used for modeling the control-flow of a system. Based on the current state and an incoming event, the actor evaluates a boolean expression and generates a token either on $q$ or on $\neg q$. Although the actor can be generalized, in this section we focus on an actor that specifically suits the C-HEAP protocol.

To model the control-flow of the C-HEAP protocol, we require a conditional branch actor that is able to evaluate whether or not a token is available in the buffer, and generate a token as an indication. Let the size of the communication memory size be some natural number $F$. The conditional branch actor $CB_{n>0}$ has two inputs $d$ and $c$ and two outputs, $q$ and $\neg q$, as depicted in Figure 4.2(a). The actor’s relation is defined by the DTA depicted in Figure 4.2(b).

The actor has an internal state $S = (n : \mathbb{N} \leq F)$. In the C-HEAP context, the state of $CB_{n>0}$ represents the number of data tokens currently available in the buffer that have not yet been communicated to the consumer. An event at the input $d$ indicates that the value of writec has been increased, representing the availability of a new data token in the buffer. An event at the output $q$ denotes that an indication of the token has been passed to the consumer. Since the state of the actor represents the number of tokens currently available in the buffer, for every input event on $d$, the state $n$ is increased, and for every output event on $q$, the state is decreased.

Upon an input event on $c$, the actor will produce a token on $q$ when the condition is true (i.e., $n > 0$) and a token on $\neg q$ otherwise. For clarity, we labeled the Büchi accepting states with a number indicating the current state of the system. The state labeled $F$ denotes a full buffer. In a real system, we do not expect that a $d$ token can arrive when the buffer is full. However, we explicitly add the transition labeled $d$ to make the actor input-complete.

Recall that in the C-HEAP protocol implementation, the status of the buffer is checked in a conditional while-loop. This actor is useful for our modeling purposes since it can be used for modeling the control-flow of a system when a branching decision is made based on some condition on the state of the system. In the next section we combine this actor with a merge actor and model the control-flow of a while-loop (a typical implementation for the claim_data primitive, as we illustrated in Listing 2 of Section 3.2). The actor $CB_{n>0}$ is deterministic. The actor is not $\subseteq$-monotonic. This will be illustrated in the context of the claim_data model in Section 4.2.3.

\[39\]
4.2.2 Model

In this section, we provide a model CD for the claim_data primitive. The primitive is called by the consumer whenever data is required and returns when data is available. In addition, in order to decide whether or not it is possible to return from the procedure, two memory locations are loaded and compared. The first is readc, a counter managed by the consumer, which is not expected to change during the execution of the primitive as the consumer is currently blocked. The second is writec, which is incremented after a new data token has been produced by the producer. The value update of the memory location of writec is the critical event in the system, enabling the return of claim_data. As such, we define the interface ports of CD as depicted in Table 4.1. The events c and r correspond to the channel events CD, c and CD, r, respectively, appearing in the channel-model of Figure 3.16. The event d is internal to the channel and corresponds to the event RD,r, denoting update of writec by release_data.

In our behavioral model we abstract away from several implementation details, but still try to capture the actual flow of the hardware and software. At the software level, we capture the essential operations performed in the claim_data primitive, which are the following:

1. Reading the value of readc
2. Reading the value of writec
3. A branching decision based on the values of readc and writec: poll again, or return from the primitive.

To capture the read operations performed in the system, we use variable delay actors, modeling the timing of the hardware components involved in the load instructions: the CPU, the NoC and the memory. In some CompSOC instances, a DMA is used for assisting the CPU in communication to external locations. We abstract away from these implementation specific details by assuming that these delays have some lower and upper time bounds which can be captured by variable delay actors. As explained in Section 3, the locations of writec and readc are implementation specific. In the model we use parametrized delay actors. For an actual system, the parameters can be replaced with the real values.

To capture the branching decision we use the conditional branch actor CB_{n>0} of Section 4.2.1. In our model, we wish to abstract away from the actual values of readc and writec. Instead, we use tokens, passed in the system. These tokens represent events in the system, and should not be confused with the data tokens of the C-HEAP protocol. A token in the model can represent the event of a data token being produced (or consumed), but can also represent any other event in the system, which is not necessarily data related. As defined earlier in Table 4.1, the actor CD has the set of input ports P_{CD} = \{c, d\}, and output ports Q_{CD} = \{r\}. The relation of the actor is defined by the composition of several basic actors presented in Sections 2.2 and 4.2.1.

Since in the C-HEAP protocol a consumer claims a single data token at once, it is impossible to receive two consecutive tokens on c without a token on q between them. We assume that an implementation complies to this rule in our analysis, and will not treat such event sequences.

The model CD of claim_data is presented in Figure 4.3. The model is a composition of several parametrized variable delay actors, modeling delays in the system, and of a merge actor G and a conditional branch actor CB_{n>0}, that together model the control-flow claim_data. The arrival of a token at the input port c models the beginning of the polling procedure in claim_data. The variable delay actor \[\Delta\{\alpha_{\text{readc}}, r, \alpha_{\text{readc}}, r\}\] models the delay of the consumer issuing a read request for the value of readc. This delay includes the CPU delay, and the delay required to transfer the read request from the CPU to the

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>Event description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>in</td>
<td>call to claim_data</td>
</tr>
<tr>
<td>d</td>
<td>in</td>
<td>The value of writec is incremented in the memory. From this point on, a read-request arriving at the memory location of writec will return the updated value</td>
</tr>
<tr>
<td>r</td>
<td>out</td>
<td>return from claim_data</td>
</tr>
</tbody>
</table>

40
Table 4.2: System specific parameters used in the CD model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Minimal/ maximal delay</th>
<th>Start event</th>
<th>End event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer processor tile C to readc memory</td>
<td>$\alpha_{\text{readc}, r}$, $\beta_{\text{readc}, r}$, $\alpha_{\text{readc}, c}$, $\beta_{\text{readc}, c}$</td>
<td>load instruction initiated</td>
<td>read request at the memory</td>
</tr>
<tr>
<td></td>
<td>minimal</td>
<td>maximal</td>
<td>minimal</td>
</tr>
<tr>
<td>Consumer processor tile C to writec memory</td>
<td>$\alpha_{\text{writec}, r}$, $\beta_{\text{writec}, r}$, $\alpha_{\text{writec}, c}$, $\beta_{\text{writec}, c}$</td>
<td>load instruction initiated</td>
<td>read request at the memory</td>
</tr>
<tr>
<td></td>
<td>minimal</td>
<td>maximal</td>
<td>minimal</td>
</tr>
</tbody>
</table>

Figure 4.3: The actor CD. A composition of basic actors is a model for claim_data.

memory storing the pointer readc. The variable delay actor $\Delta_{[0,\infty]}$ models the delay required for the memory to process the read request, and the delay required to transfer the completion back to the CPU. Similarly, the variable delay actors $\Delta_{[0,\infty]}$ and $\Delta_{[0,\infty]}$ model the delays of a read request from the memory location of writec. The actor $\text{CB}_{n>0}$ models the branching decision made in the system based on the value of writec and readc. Recall that the system implementation will make a branching decision based on the difference between the counters, indicating whether or not data is available in the FIFO. The actor $\text{CB}_{n>0}$ models this branching decision by generating a token at the output of $\text{CB}_{n>0}$ if currently the state of its buffer is larger than 0. Otherwise, it generates a token on $\text{CB}_{n>0}$.q, eventually merged by the actor $G$, to create another c token. This models the behavior of the while-loop control-flow of the polling procedure.

Notice the location of the $\text{CB}_{n>0}$ actor in the composition. The moment of choice in this case is the time by which the value of writec has been updated in the memory. So although the actual evaluation of the condition in the conditional branch in the system is in fact done only after the value was loaded to the CPU (after experiencing also the delay of the completion from writec to C), the moment of choice is located before the completion has been sent, and that is where the CB actor is located. The delay from writec to requester C will still be experienced by the system for completing the load instruction, which is modeled in the form of a variable delay actor $\Delta_{[0,\infty]}$ (on both c and ¬c).

The delays in the system are abstracted to parametrized variable delay actors. The variable delay actor is general enough to model any delay, given the existence of the actor $\Delta_{[0,\infty]}$. However, we assume that for the class of predictable hardware and software systems that we model (e.g., CompSOC), there does exist an upper bound for the delay a token can experience. This is a reasonable assumption for practical systems, where the number of tokens that can be temporarily processed or buffered is finite and bounded. The variable delay actors are abstractions of the processing time and NoC delay in the system. The actual parameter values to be used in these can vary based on the implementation of the system.
It is interesting to note that the values for the minimal and maximal delays can either be accurate (the actual minimal or maximal delays the tokens may experience), or, for our purposes, can be an over-approximation. This claim is justified by Lemma 3.

**Lemma 3.** Let $A = (P,Q,R_A)$ and $A' = (P,Q,R_A')$ be two (possibly nondeterministic) actors such that $in_A \subseteq in_{A'}$, and $\forall x \in in_A, \forall y : xAy \implies xA'y$. Then it holds that:

1. $A \subseteq A'$, and
2. Let $S$ be an actor composition, and let $S'$ be the composition $S$ where all instances of $A$ are substituted by $A'$. Then it holds that $S \subseteq S'$.

**Proof.** Since $in_A \subseteq in_{A'}$, to show that $A \subseteq A'$, it remains to show that $\forall x \in in_A, \forall y : xAy \implies \exists y' : y \subseteq y' \land xA'y'$. Since for any sequence $y$, $y \subseteq y$, we can take $y$ as the witness for the abstraction.

For the same reason, the actors composition $S \subseteq S'$: for any given input sequence $x$ and output sequence $y$ such that $xSy$, the actor composition $S'$ can mimic the behavior of the actors of $S$, so it must hold that $xS'y$. Again, since $y \subseteq y$, we can take $y$ as the witness for the abstraction and we can conclude that $S \subseteq S'$.

According to Lemma 3, for any two variable delay actors $\Delta_{[\alpha_1,\beta_1]}$ and $\Delta_{[\alpha_2,\beta_2]}$ such that $\alpha_2 \leq \alpha_1 \land \beta_1 \leq \beta_2$, we can conclude that $\Delta_{[\alpha_1,\beta_1]} \subseteq \Delta_{[\alpha_2,\beta_2]}$. Furthermore, let $S$ be an actor composition, and let $S'$ be the composition $S$ where all instances of $\Delta_{[\alpha_1,\beta_1]}$ are replaced by $\Delta_{[\alpha_2,\beta_2]}$, then $S \subseteq S'$. Notice that for the abstraction relation $\Delta_{[\alpha_1,\beta_1]} \subseteq \Delta_{[\alpha_2,\beta_2]}$ to hold, it is sufficient that the condition $\beta_1 \leq \beta_2$ holds, without any restrictions on the relation between $\alpha_1$ and $\alpha_2$. However, to substitute the actor $\Delta_{[\alpha_1,\beta_1]}$ by its abstraction within a context $S$, in the general case we also require that $\alpha_2 \leq \alpha_1$. This requirement is necessary if $S$ adheres to non-monotonic behavior (i.e., the composition includes a non-monotonic component).

Lemma 3 is useful because it allows us to make over-approximations of component delays and achieve sound WC performance analysis, as it implies we can replace a component with a variable delay actor $\Delta_{[\alpha,\beta]}$, as long as $\alpha$ is less than or equal to the best-case delay of the component, and $\beta$ is greater or equal to the WC delay of the component. In some cases the components have existing conservative models (e.g., the NoC model in CompSOC [11]). As such, these may not necessarily include the best-case delays of a component. It is still desired to use these in non-monotonic contexts, rather than remodel the component. In this case, it is possible to use the variable delay actor $\Delta_{[0,\beta]}$, where $\beta$ is the WC delay of the component according to the existing conservative model. It could be claimed that using the under-approximation 0 is over-conservative, since the best-case delay could be much higher. In Section 4 we see that for the case of our models, in the SDF abstraction, the minimal delay is no longer observable, and as such the choice to take $\Delta_{[0,\beta]}$ is not problematic.

So far we discussed the model for the `claim_data` primitive. The model for `claim_space` is symmetrical and appears in Figure 4.4.

![Figure 4.4: Model for claim_space](image)

### 4.2.3 Non-monotonicity of claim_data

The actor CD models the polling procedure initiated by the consumer in the `claim_data` primitive. As was already observed in [15], a polling procedure is non-monotonic. In this section we show that the
non-monotonicity of CD can be indeed be observed in our model.

In claim_data, the initiator of the polling procedure iteratively loads certain values (readc, writec) from memory, and evaluates a condition based on these values. Each loading instruction has a (possibly variable) delay. Assume the moment at which the polled writec is updated to indicate the availability of a data token is fixed and is equal to \( t_d \). The time at which a read request arrives at the memory containing writec is denoted by \( t_c \) and depends on the response-time of various components (modeled as the variable delay actors in Figure 4.3). For a fixed \( t_d \), the earliest possible finishing-time of CD (i.e., a generation of a token on \( r \)) is encountered when the memory request happens just after the memory was updated, i.e., \( t_c = t_d - \delta \), where \( \delta \) is an infinitesimally small constant. In contrast, if the response-time is better (i.e., shorter), and the request arrives at the memory just before the memory is updated (i.e., \( t_c = t_d - \delta \)), then an additional polling iteration is required, and the finishing-time of CD is worse (i.e., a token on \( r \) is generated later). This example of non-monotonicity in CD is formalized by the following lemma.

**Lemma 4.** The actor CD with finite variable delay actors is not \( \sqsubseteq \)-monotonic.

**Proof.** Recalling the definition of \( \sqsubseteq \)-monotonicity presented in Section 2, in order for CD to be \( \sqsubseteq \)-monotonic, we require that for every \( x, y \) and \( x' \) such that \( x, y \in in_{CD} \) and \( x \subseteq x' \), exists a \( y' \) such that \( y \sqsubseteq y' \). We provide a counterexample that can be constructed for any set of finite variable delay actors. For brevity, we compose the variable delay actors \( \Delta_{\{a_{\text{readc}},c\}} \Delta_{\{a_{\text{writec}},r\}} \Delta_{\{a_{\text{writec}},c\}} \Delta_{\{a_{\text{writec}},c\}} \) into a single variable delay actor \( \Delta_{\{a_{\text{readc}},c\}} \) (the serial composition of variable delay actors is again a variable delay actor). To keep the notation consistent, we also rename the actor \( \Delta_{\{a_{\text{readc}},c\}} \) to \( \Delta_{\{a_{\text{cmp}},\beta_{\text{cmp}}\}} \).

This transformation results in the model appearing in Figure 4.5.

![Figure 4.5: The claim_data model](image_url)

We need to show that there exist \( x, y \) and \( x' \), such that \( x' \in in_{CD} \) and \( x \subseteq x' \) but there does not exist a \( y' \) such that \( x' \sqsubseteq y' \). We explain how to construct the traces \( x, y \) and \( x' \) for any parameters of the variable delay actors, illustrated by Figure 4.6.

Consider the input trace \( x = \{(c,(t_d - \beta_{\text{req}} - \delta) \cdot \epsilon),(d,t_d \cdot \epsilon)\} \), where \( t_d \) is the time of the event at input port \( d \) and can be chosen arbitrarily, and \( \delta \) is some small constant. The input trace consists of a single input token on port \( c \) and a single input token on \( d \). The input trace \( x \) is illustrated in Figure 4.6(a). We use the small constant \( \delta \) to avoid handling the case where both input events on \( CB_{n \geq 0} \) occur at the same time. For our chosen input sequence \( x \), we relate an output sequence \( y \) in the following manner, as illustrated by Figure 4.6(a).

A token is available at the input \( d \) at time \( t_d \) (1). Due to the token on \( c \) at time \( t_d - \beta_{\text{req}} - \delta \) (2), a token is generated on \( G, q \) after \( d_{\text{max}} \) time, at time \( t_d - \delta \) (3). The token is available at \( CB_{n \geq 0}, c \) (4) just before the arrival of the token, generating another iteration of polling (5-7). The next token on \( CB_{n > 0}, c \) will arrive at \( t_d - \delta + \beta_{\text{cmp}} + \beta_{\text{req}} \) (8). By this time, the token \( x(d)(0) \) is already available at \( CB_{n \geq 0}, d \), hence a token will be generated on \( CB_{n > 0}, q \) at time \( t_d - \delta + \beta_{\text{cmp}} + \beta_{\text{req}} \) (9). Following the token on \( CB_{n > 0}, q \), a token arrives at the output \( r \) at time \( t_d - \delta + \beta_{\text{cmp}} + \beta_{\text{req}} + \beta_{\text{cmp}} = t_d - \delta + \beta_{\text{req}} + 2\beta_{\text{cmp}} \) (10). The output \( y \) related to \( x \) by CD is thus \( y = \{r,(t_d - \delta + \beta_{\text{req}} + 2\beta_{\text{cmp}} \cdot \epsilon)\} \).

Now, we construct a sequence \( x' \) such that \( x' \in in_{CD} \) and \( x \subseteq x' \). We keep the timing of the data token arrival the same: \( x'(d)(0) = x(d)(0) = t_d \), and choose the timing of \( x'(c)(0) \) to be \( t_d - \alpha_{\text{req}} + \delta \). The input trace \( x' = \{(c,(t_d - \alpha_{\text{req}} + \delta) \cdot \epsilon),(d,t_d \cdot \epsilon)\} \) is illustrated in Figure 4.6(b). The input trace indeed satisfies that \( x \sqsubseteq x' \). However, there does not exist a \( y' \) such that \( x' \sqsubseteq y' \). By taking the time of the input
token as we do, we ensure that an event is available at $\text{CB}_{n>0}.c$ (4) after the data token has arrived on $\text{CB}_{n>0}.d$ (1), without requiring a second iteration of polling. The maximal time the output token could arrive at $r$ (6) is thus $t_d - \alpha_{\text{req}} - \delta + \beta_{\text{req}} + \beta_{\text{cmp}}$.

Recall that we want to show that there does not exist a $y'$ such that $x' \text{CD} y'$ and $y \subseteq y'$. Any output sequence related to $x'$ by CD will consist of a single token on $r$. To show that for a $y'$ such that $x' \text{CD} y'$ it cannot hold that $y \subseteq y'$, it suffices to show that $y(r)(0) > y'(r)(0)$. The maximal time at which $y'(r)(0)$ can occur is $t_d + (\beta_{\text{req}} - \alpha_{\text{req}}) + \delta + \beta_{\text{cmp}}$. As we chose $y(r)(0) = t_d + 2\beta_{\text{req}} + 2\beta_{\text{cmp}}$, assuming that $\delta$ is infinitesimally small, it must hold that $t_d + (\beta_{\text{req}} - \alpha_{\text{req}}) + \delta + \beta_{\text{cmp}} \leq t_d + 2\beta_{\text{req}} + 2\beta_{\text{cmp}}$, and thus $y(r)(0) > y'(r)(0)$. 

The non-monotonicity of the actor CD implies that simply composing conservative models of the individual components of the system may not necessarily result in a conservative model [15]. We illustrate this in Figure 4.7. Figure 4.7 sketches the time of the generation of the $n$-th token on $r$ as a function of the arrival-time of the $n$-th token on $c$, for a fixed arrival-time of $d$. Notice that the actor CD
is non-deterministic, and Figure 4.7 is a mere illustration showing a possible trend.

Depending on the time of $c(n)$, the earliest possible production time of $r$ is the time of the arrival of a token on $d$, with the addition of the minimal completion-time (i.e., $d(n) + \alpha_{cmp}$). We also mark the time $d(n) + \beta_{req} + 2\beta_{cmp}$ as this time effects the structure of the graph. If $c(n)$ arrives before $d(n)$, this is the finishing-time of CD (we discuss this point again in Section 4.2.4). Figure 4.7 illustrates the effect of the non-monotonicity of CD. Assume that the WC arrival-time of $c$ is as illustrated by Figure 4.7. If one only considers the time of $r(n)$ that corresponds to the WC arrival-time, then the actual WC finishing-time is in fact hidden. In fact, a whole range of worse finishing-times exist, illustrated by the highlighted region in Figure 4.7.

![Figure 4.7: Intuitive illustration: claim_data is non-monotonic. The response-time of the output $r(n)$ as a function of the input $c(n)$, for a fixed $d(n)$.](image)

For the final goal of creating an SDF model of the system, we need to specifically treat this non-monotonic components and provide an abstraction for it. We try to accomplish this in the next section.

### 4.2.4 Abstraction

The model of the claim_data primitive presented in the previous section is non-deterministic and non-monotonic. As explained in Section 3, to achieve a predictable platform, suitable for real-time applications, we require conservative models. We choose to use a conservative SDF model, so that it is also easily integrated with the current modeling scheme of CompSOC. In this section we present an SDF model that is an abstraction of the CD actor presented in Section 4.2.2, and provide a proof that the abstraction relation holds. Since CD is input-complete, $\preceq$-monotone, and $\preceq$-continuous, providing an SDF abstraction enables using our abstraction framework of Section 2.6.

A conservative SDF model of the primitive claim_data is presented in Figure 4.8.

![Figure 4.8: The actor CD_{SDF}: an SDF model for claim_data.](image)

**Lemma 5.** The SDF model $CD_{SDF}$ is an abstraction of model $CD$: $CD \sqsubseteq CD_{SDF}$

**Proof.** To prove the abstraction/refinement relation we must show:

1. $in_{CD_{SDF}} \subseteq in_{CD}$. The actor $CD_{SDF}$ is input-complete, so this must hold.
2. $\forall x \in in_{CD}, \forall y: xCDy \Rightarrow \exists z: xCD_{SDF}z \land y \sqsubseteq z$. For our case, the actor $CD_{SDF}$ is deterministic, so in fact we need to prove that $\forall x \in in_{CD}, \forall y: xCDy$, let $z$ be the output trace related to $x$ by the relation of actor $CD_{SDF}$, then $y \sqsubseteq z$. For both actors, the length of the output sequence on port $r$ is equal to $l = min(|x(c)|, |x(d)|)$. According to the semantics of an SDF actor explained in Section 2, forall
n < \lfloor y(r) \rfloor$, it holds that $y(r)(n) = \max(x(c)(n), x(d)(n)) + \beta_{req} + 2\beta_{cmp}$. To prove refinement, we show that for actor CD, for all $n < l$ it holds that $y(r)(n) \leq \max(x(c)(n), x(d)(n)) + \beta_{req} + 2\beta_{cmp}$.

We distinguish between two cases:

- $\max(x(c)(n), x(d)(n)) = x(c)(n)$. In this case, the data token arrives at CB$_{n>0}$ and is buffered until a synchronization token arrives. It holds that $y(r)(n) \leq x(c)(n) + \beta_{req} + 2\beta_{cmp}$.

- $\max(x(c)(n), x(d)(n)) = x(d)(n)$. This implies that $x(c)(n) \leq x(d)(n)$. After the arrival of the token $x(c)(n)$, a token will be generated at $G.q$, followed by a token at CB$_{n<0..c}$ at $t \in [x(c)(n) + \alpha_{req}, x(c)(n) + \beta_{req}]$. If by $x(d)(n) < t$, then the $n$-th token will be generated on $q$ and $y(r)(n) \leq x(d)(n) + \beta_{req} + \beta_{cmp}$. Otherwise, a token will be generated on CB$_{n<0..q}$, initiating a periodic polling with an intermediate arrival time of a token on CB$_{n<0..c}$ at most $\beta_{req} + \beta_{cmp}$. This implies that $n$-th token arriving at $d$ can be delayed by at most $\beta_{req} + \beta_{cmp}$ before CB$_{n<0..d}$ will be generated. Finally, CB$_{n<0..d}(n) \leq x(d)(n) + \beta_{req} + \beta_{cmp}$ implies a bound for the $n$-th token on port $r$: $y(r)(n) \leq x(d)(n) + \beta_{req} + 2\beta_{cmp}$.

\[ \square \]

In this chapter we provided a conservative SDF abstraction to the claiming primitives of C-HEAP. The SDF delays are parametrized. In the next section we explain how to extract values for the parameters from existing models of the system resources. In addition, we provide models for the remaining primitives of the communication-channel.
Chapter 5

Channel-Model

In this chapter we model the remaining components in the channel-model (i.e., Read, Write, and the releasing primitives of C-HEAP), which are all monotonic, hence are directly modeled as SDF graphs. In addition, we explain how to assign values to the parameters of the SDF model derived for the claiming primitives in Chapter 4. We relate all models to the execution model of Figure 3.16. In addition, we use the shared-buffer model of Section 3.5. Modeling a shared-buffer becomes necessary even if we assume no interference between tasks. This is since both the Write operation and release data share the same HW path (i.e., the DMA, NoC, and memory). This implies potential interference between the operations that needs to be modeled. Throughout the models we use several common notations for some system parameters, summarized in Table 5.1.

Table 5.1: Common notations used throughout the primitive models

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer/queue sizes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B_P$</td>
<td>[T]</td>
<td>size of local buffer at the processor tile of the source actor</td>
</tr>
<tr>
<td>$B_{NI}$</td>
<td>[flit]</td>
<td>network-interface buffer size</td>
</tr>
<tr>
<td>$B_M$</td>
<td>[MSU]</td>
<td>Memory buffer size</td>
</tr>
<tr>
<td>$R$</td>
<td>[1]</td>
<td>maximal number of pending DMA requests</td>
</tr>
<tr>
<td>Rates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r$</td>
<td>[T]</td>
<td>number of tokens produced by a producing source actor in a single firing</td>
</tr>
<tr>
<td>$b$</td>
<td>[T/DMABS]</td>
<td>number of DMA burst required for a single data token</td>
</tr>
<tr>
<td>$d$</td>
<td>[DMABS/flit]</td>
<td>number of NI-slots required for a single DMA burst</td>
</tr>
<tr>
<td>$m$</td>
<td>[MSU/flit]</td>
<td>number of flits in a memory service unit</td>
</tr>
</tbody>
</table>

5.1 Read data

Figure 5.1 provides a model for a read access performed by a consumer-task $C$ in order to fetch data from a remote memory $M$. To show the composition of the model with the SDF execution model of Section 3.6, the model includes the dotted actors $C$.Read($i$) and $C$.CD($i+1$). The index $i$ indicates that we model a read access performed on the $i$’s communication-channel of task $C$. Recall that after reading is completed, the task will claim data on channel $i+1$, hence we include also the actor $C$.CD($i+1$). All interactions (i.e., SDF channels) of the Read model and the execution model appear in Figure 5.1.

A read access is blocking, i.e., after it has begun, execution of the following instructions of the application on the CPU will only begin after the read access has completed. As such, no new memory requests can be issued during the read transaction. Since instructions are executed sequentially, no interference between tasks of the same application exists. In Figure 5.1 we model the read transaction as a composition of the models of the HW resources involved in the transaction:
5.2 claim_data and claim_space parameters

In Section 4.2 we provided an SDF model $CD_{SDF}$ for claim_data. The model consisted of a single SDF actor with a parametrized execution time. Next, we explain how to derive values for the model parameters, provided that we have existing conservative models of the HW components involved in the claim_data primitive. We explain how to extract the parameters for claim_data, and the exact same method can be used for claim_space. Recall that $CD_{SDF}$ is a composition of various actors, where variable delay actors model the delays of read requests and read completions of the polling procedure. The parameters we need to assign values to are:

1. $\beta_{req}$ - the maximal latency between the time at which the consumer CPU initiates a read request to $writec$ and the time at which the request is executed (i.e., serviced) by the memory controller.

2. $\beta_{cmp}$ - The maximal latency between the time at which the request starts being serviced by the memory and until the time at which a completion has reached and processed by the consumer CPU.

We conservatively model these delays using a composition of SDF models, similar to that used in the conservative model of a read transaction of Figure 5.1. The models for $\beta_{req}$ and $\beta_{cmp}$ are depicted in Figure 5.2.

We assume that the memory requirements for storing $readc$ or $writec$ are small enough such that they fit within a single memory service unit. As such, the memory rate actor $M_{pc}$ has no self-edge, since only a single execution of the actor is needed, and since read requests for $writec$ arrive sequentially (i.e., one at a time).

In Figure 5.2(a) we model the path from the generation of a read request on the CPU and until the memory controller starts providing service to the request. The sequence of actors includes an actor for
the DMA latency, the NoC and the memory latencies. Recall that the actor modeling the DMA has no self-edge since read requests are blocking, hence are executed one at a time. Also notice that since we model a read request, as we assume a read request requires only a single flit on the NoC, the NoC rate actor \( N_{\rho CM} \) has no self-edge. We model the latency between the time at which a request arrived at the memory interface, and until the time at which the memory started providing service to the request using the latency-actor \( M_{\theta C} \) of the latency-rate memory model. As such, \( \beta_{\text{req}} \) is conservatively bound by the sum of the execution-times of the sequence of actors appearing in Figure 5.2(a).

Figure 5.2(b) models the path of the transaction’s completion. The path begins with the memory model, modeling the generation of the completion. Recall that as mentioned in Section 3.4.3, the memory controller always delays the response such that the actual response-time is equal to the response-time of the latency-rate model (i.e., not better). As such, to conservatively model the delay of the memory, we include both the latency- and the rate-actors. From the memory, the completion is passed on to the NoC and collected at DMA on the tile side. In the model we add an actor to model the response-time of the actual processing of the data read from the memory.

Figure 5.3 shows the composition of the SDF actor modeling claim_data with the corresponding actors in the task execution model. We explicitly mark the ports of the actor: d, c and r. Similarly, the composition of the model for claim_space is provided in Figure 5.4.

5.3 Writing and releasing data

In this section we model a write operation and the primitive RD. We model these in the same section for the following reasons. First, releasing data is essentially a write access, hence the models are very similar. Second, and more importantly, in some system configurations, interference exists between writing data and releasing it. The easiest way to model the interference is to model both actions together.

We distinguish between two possible flavors for a write access, as explained in Section 3.2.1:
1. Non-blocking write - the CPU resumes execution of the instruction following the write access immediately after issuing a store instruction.

2. Blocking write - the CPU resumes execution only after ensuring the data has been written to memory. Effectively, a blocking write is a non-blocking write augmented with a read access.

**Blocking write access**

Figure 5.5 depicts the model of a blocking write access performed by task $P$ on its $j$’s C-HEAP channel. Each row models the following operation:

1. The top row models the path of a write access.
2. The middle row models a read request from the CPU to the memory, and
3. The third row models the completion from the memory to the CPU.

In Section 3.4.4 we provide an example of a composition modeling a write access. The top row of the model is similar, but not identical. The write operation is modeled by the following sequence of actors:

- **buf** - models the local buffer storing the data to be written. We assume the local buffer is large enough to fit a single firing of the task. Since write accesses are performed sequentially, no back-pressure from the buffer exists in the model (unlike the composition example of Section 3.4.4).
- **DMA write** - models the delay required for the DMA to pass the data to the NoC. A single data token requires $b$ DMA bursts, hence the rates of the input and output ports of the channel (buf, DMA write).
• \( N_{\theta_{PM}} \) and \( N_{\rho_{PM}} \) - model the write request on the NoC path from the producer tile to the memory. A single DMA burst requires \( d \) flits on the NoC. The back-pressure from the network-interface buffer is modeled by the channel \( (N_{\rho_{PM}}, \text{DMA write}) \) with an initial number of tokens \( B_{NI} \) equal to the number of flits in the network-interface buffer.

• \( M_{\theta_{P}} \) and \( M_{\rho_{P}} \) - model the response-time of the memory. A single memory service unit equals to \( m \) flits. This is modeled by the rates on the input and output ports of channels \( (N_{\rho_{PM}}, \text{M write}) \) and on the back-pressure channel \( (M_{pp}, N_{\theta_{PM}}) \). The initial number of tokens \( BM \) on \( (M_{\rho_{P}}, N_{\theta_{P}}) \) models the size of the memory input buffer.

The second and third rows model a read request and a completion, respectively. The actors model the same path modeled in the Read model of Figure 5.1. The read request is only performed in order to ensure that the preceding write has completed. As such, the size of the read request is minimal, and we assume all actors in the path require a single execution (e.g., the rate-actors of the NoC and the memory). For this reason, all rates on the edges are 1.

The write and read operations include interference, which is modeled by the vertical channels from the top row to the middle row. The read request can only begin being served by the DMA, the NoC, or the memory after the write request has been handled. This restriction only applied for the actors modeling components that must execute sequentially (i.e., have self-edges). Since we know the number of executions required by each of the actors modeling these components, we model the interference by adding a channel from the actor in the write path to the corresponding actor in the read path. The port rates represent the number of executions performed by the write actor before the read request can be serviced. For example, we draw an edge between DMA write and DMA read. The number of DMA executions that are a result of the write access is \( r \cdot b \). Only after these have been completed, the read request can be handled by the DMA. We model this using a channel (DMA write, DMA read) with a rate of \( r \cdot b \) DMA write firings for a single DMA read firing. Similarly, we add the channels \( (N_{\rho_{PM}}, N'_{\rho_{PM}}) \) and \( (M_{pp}, M'_{pp}) \). Notice that in the model we assume that \( r \) is a natural number. If this is not the case, we require that \( r \) is rounded-up to \( m \cdot \lceil \frac{r}{m} \rceil \).

The complementary model for release data is depicted in Figure 5.8. Recall that the reason for performing a blocking-write is that the data and the counter are located in different memories. As such, for the case of a blocking write, the writing operation and release data are performed on different NoC connections and on different memories, hence do not share any resources. Since no interference exists, we simply model release data as the write access of Figure 5.8.

![Figure 5.6: release data after a blocking write](image_url)

**Non-blocking write access**

Unlike the case of a blocking write access, for the case of a non-blocking write, interference does exist between writing the data and releasing it. The interference results in the slightly more complex model of Figure 5.7. The DMA, the NoC and the memory are shared resources, and are modeled using the technique presented in Section 3.5. The order of execution of writing and releasing is statically ordered: in each firing of a task, the task always first writes data, and then releases. This enables using the technique of Section 3.5. Notice that no interference exists on the local buffer, modeled by the actor buf. We assume that writec has a dedicated memory location (recall that in fact writec is simply a pointer to a memory location).

The top row models the path of the write access for writing the data token. The bottom row models releasing the data, i.e., writing writec. The tagged actors model shared components which are used by the releasing primitive. To model the temporal interference between shared resources, we add several auxiliary c actors, orchestrating the order of firings of actors providing service to both the writing and the releasing of data, as explained in Section 3.5. To model the spatial interference between the write and
release operations, we connect back-pressure channels from each resource back to both the tagged and untagged actors requiring that resource, as explained in detail in Section 3.5. The DMA has a limited request-queue, hence required an addition auxiliary actor $h$ to collect the $b$ firings and convert them back into a single request. The number of initial tokens on the edges can be calculated as explained in Section 3.5.

Notice that in the case of a non-blocking write access, for the DMA we model the spatial interference of a shared request-queue, and for the NoC and the memory, we model spatial interference of a shared buffer. In Section 3.5 we mention that interference between tasks on the NI buffer (i.e., the NoC buffer) cannot occur in CompSOC as sharing of NI ports is not allowed. However, here interference in fact exists both on the NoC and on the memory since both the write operation and release_data are activated by the same task.
Figure 5.7: Non-blocking write and release data
Release space

Releasing space does not involve writing of actual data to memory, as such no interference needs to be modeled. As such, the model for release space is similar to that of release data in the blocking-write configuration.

Figure 5.8: release data after a blocking write
Chapter 6

Related work

Compositional non-monotonic abstraction

Our framework for abstraction is mainly based on TAI [8], where a general formalism for timed actors is developed and several useful propositions are provided to allow the abstraction of time-non-deterministic systems into deterministic ones. The authors target the work for allowing performance-preserving transitions between formalisms. Our work can be seen as a refinement of this work. We both add additional general propositions, and direct the theory towards the specific purpose of abstraction of non-monotonic components into SDF graphs.

Our work is focused on the ultimate goal of achieving conservative SDF models that can be integrated into existing modeling and analysis frameworks. A common alternative technique for deriving worst-case performance bounds exists in the RT calculus [23] or network calculus [13]. In these, the worst-case (or best-case) behavior of components is typically expressed as arrival and service curves. These curves express the cumulative number of events at the input or output of a component. As such, the curves have monotonicity naturally embedded in them. If non-monotonicity exists in the system, it thus somehow has to be abstracted before being expressed as a service or arrival curve. In this sense, our work can be seen as complementary also in this field.

Monotonicity enables considering only the worst-case temporal behavior of individual components in a composite system. In case non-monotonic behavior exists in a system, model-checking is an alternative to component abstraction. The system can be described as timed-automata [2] and model-checked for temporal properties (e.g., using [12]). However, the benefits of modeling a system using a time-deterministic model no longer exist. As such, the models could suffer from a large state-space, increasing the complexity of the analysis, and potentially result in state-space-explosion.

Modeling of Static-order schedules

SDF modeling of static-order schedules has been recently studied in [6], where the authors provide a technique to model static-order schedules directly on an SDF graph. This is in contrast to the method suggested in [3], which requires conversion of the SDF graph to a homogeneous-SDF graph (an SDF graph with all rates equal to 1). Avoiding this conversion is beneficial as it reduces the number of actors in the graph, and maintains its original structure. Our modeling of a shared buffer in also in the field of static-order schedules, and can be applied orthogonally to the SDF graph generated by the technique in [6], in case the scheduled tasks in the graph share a buffer.
Chapter 7

Conclusions

In this thesis we presented a conservative model for C-HEAP, the intertask communication-channel used in the CompSOC platform. At a higher level, we aim at using this case-study to understand non-monotonic behavior and explore the possibilities for conservative abstraction of non-monotonic systems.

Throughout the modeling process, we introduced several methods and techniques which can also be applied to general-purpose modeling. We list our contributions below.

- An abstraction-framework for conservative modeling of non-monotonic components. Our abstraction is compositional, i.e., it allows treating individual components separately, and then composing them together. The composition is performed such that the worst-case performance guarantees derived for the abstract composite-model are still valid for the refined one. We apply the framework on C-HEAP’s (non-monotonic) polling procedures and derive an SDF abstraction, which conveniently fits within the CompSOC framework.

- A refined and revised communication-channel model for the binding-aware graph used in the performance analysis in the CompSOC tool-flow. Deriving a communication-channel required structurally defining the components of the channel, the interaction between them, and the interaction of the channel with the task’s execution.

  Our refinement of existing models allows tighter analysis of worst-case temporal behavior. Our model is a revised one as it now models potential interference between tasks of the same application running on the same processor. This interference was not modeled in previous instances.

- Composing the channel models required extensive exploration and documentation of the existing model compositions performed in the CompSOC tool-flow. In addition, throughout this process an implementation bug was discovered in the platform.

- A modeling technique for buffer sharing between tasks under a static-order schedule. The ability to model such shared resources is essential for modeling the DMA HW block in CompSOC. Our modeling also enables modeling of shared NoC ports, which is currently not allowed. Our models enable such sharing in future configurations of the platform.

Our aim in this work was two-fold. First, we were interested in deriving a conservative C-HEAP communication-channel model. Second, we were interested in using our case-study as a basis for better understanding of non-monotonic behavior and its abstraction.

The first goal was successfully achieved, and also brought rise to additional modeling contributions that can be used also outside the scope of the CompSOC platform. The test-case also showed that the notions raised in TAI are useful and can be applied in practice. Our second goal was found to be beyond the extent of this work, and is part of the future work which is listed next.

7.1 Future work

In this work we studied a certain instance of non-monotonic behavior, namely, a polling procedure. Throughout the work, several ideas regarding the nature of non-monotonic behavior were raised, but due to the limited scope of the work, they were not properly consolidated and their investigation is left for future work. As such, we list the items that we feel should be further investigated in this field.
• We believe that further study on the typical hardware and software systems that could possibly adhere to non-monotonic behavior is still required. Classification of systems could assist the system modeler in the detection of non-monotonic behavior.

• In addition, it is interesting to characterize non-monotonic components in order to better understand one’s ability to (possibly automatically) abstract them into SDF graphs.

• In our TAI models we used a conditional branch actor \( \text{CB}_{n \geq 0} \) to model the while-loop of the polling procedure. We believe that this actor can be generalized to a conditional-branch actor with a more general branch condition. Such an actor can be useful in modeling of while-loops and conditional-branching in TAI.

In addition, whereas this work precisely defines how the communication-channel models should be embedded in the binding-aware graph of the CompSOC tool-flow, the work is still to be integrated into the tools.
Bibliography


