



Modélisation des systèmes synchrones en BIP

Vasiliki Sfyrla

► To cite this version:

Vasiliki Sfyrla. Modélisation des systèmes synchrones en BIP. Autre [cs.OH]. Université de Grenoble, 2011. Français. <NNT : 2011GREN022>. <tel-00688253>

HAL Id: tel-00688253

<https://tel.archives-ouvertes.fr/tel-00688253>

Submitted on 17 Apr 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Vasiliki SFYRLA

Thèse dirigée par **Joseph SIFAKIS**

et codirigée par **Marius BOZGA**

préparée au sein du **Laboratoire VERIMAG**

et de l'**Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Modélisation des Systèmes Synchrones sur BIP

Thèse soutenue publiquement le **21 Juin 2011**,
devant le jury composé de :

Mr. Nicolas HALBWACHS

Directeur de Recherche au CNRS, VERIMAG/CNRS, Président

Mr. Albert BENVENISTE

Directeur de Recherche à l'INRIA, IRISA/INRIA, Rapporteur

Mr. Jordi CORTADELLA

Professor, Université Polytechnique de Catalogne (UPC), Rapporteur

Mr. Daniel KROENING

Fellow, Magdalen College, Examineur

Mr. Roberto PASSERONE

Assistant Professor, Université de Trente, Examineur

Mr. Joseph SIFAKIS

Directeur de Recherche au CNRS, VERIMAG/CNRS, Directeur de thèse

Mr. Marius BOZGA

Ingénieur de Recherche au CNRS, VERIMAG/CNRS, Co-Directeur de thèse



Table des matières

Abstract	5
1 Introduction	9
2 The BIP Framework	15
2.1 Abstract Model of BIP	16
2.1.1 Behavior	16
2.1.2 Abstract Model of Interactions	16
2.1.3 Abstract Model of Priorities	17
2.1.4 Composition of Atomic Components	17
2.2 Concrete Model of BIP	18
2.2.1 Modeling BIP Atomic Components	18
2.2.2 Semantics of Atomic Components	20
2.2.3 Concrete Model of Interactions	23
2.2.4 Concrete Model of Priorities	24
2.2.5 Composition of Atomic Components	24
2.3 The BIP Language	27
2.4 The BIP Toolset	30
2.4.1 Code Generation for BIP Models	32
2.5 Discussion	33
3 Synchronous Formalisms	37
3.1 The LUSTRE Language	37
3.1.1 Single-clock Operators	38
3.1.2 Multi-clock Operators	39
3.1.3 LUSTRE Compiler and Code Generation	41
3.2 SIGNAL	42
3.2.1 Clock Relations	43
3.2.2 Single-clocked operators	43
3.2.3 Multi-clocked operators	44
3.2.4 Parallel Composition	44
3.2.5 An example	44
3.3 MATLAB/Simulink	45
3.3.1 Signals	45
3.3.2 Ports and Atomic Blocks	46
3.3.3 Subsystems	47
3.4 Discussion	49

4	Modeling synchronous data-flow systems in BIP	53
4.1	Cyclic BIP Components	54
4.1.1	Modeling Cyclic BIP Atomic Components	54
4.1.2	Composition of Cyclic BIP Atomic Components	56
4.2	Synchronous BIP Components	59
4.2.1	Modeling Synchronous BIP Atomic Components	59
4.2.2	Well-triggered Modal Flow Graphs	60
4.2.3	Composition of Synchronous BIP Atomic Components	64
4.3	Structural Properties of Synchronous BIP Components	67
4.4	The Synchronous BIP Language	69
4.5	Related Work	71
4.6	Conclusion	73
5	Language Factory for Synchronous BIP	75
5.1	From LUSTRE to Synchronous BIP	75
5.1.1	Principles of the Translation for Single-clock LUSTRE Nodes	75
5.1.2	Translation of Single-clock LUSTRE Operators	76
5.1.3	Principles of the Translation for Multi-clock LUSTRE Nodes	78
5.1.4	Translation of Multi-clock LUSTRE Operators	83
5.1.5	Implementation of the Translation	89
5.2	From MATLAB/Simulink into Synchronous BIP	89
5.2.1	Principles of the Translation	89
5.2.2	Translation of Simulink Ports and Simulink Atomic Blocks	90
5.2.3	Translation of Triggered Subsystems	92
5.2.4	Translation of Enabled Subsystems	95
5.2.5	Clock Generator	97
5.2.6	Translation of a Simulink Model	100
5.2.7	Implementation of the Translation	100
5.2.8	Similar Translations	100
5.3	Conclusion	101
6	Code Generation for Synchronous BIP	103
6.1	Sequential Implementation	103
6.1.1	Experimental Results	104
6.2	Distributed Implementation	106
6.2.1	Direct Method for Distributed Code Generation	107
6.2.2	Cluster-oriented Method for Distributed Code Generation	112
6.3	Related Work	115
6.4	Discussion	116
7	Representation of Latency-Insensitive Designs in Synchronous BIP	117
7.1	The Methodology	117
7.2	The Single-clock Synchronous Design	119
7.3	Transformation of Synchronous BIP Systems to LID	121
7.4	Discussion	126
8	Conclusion	127

Abstract

A central idea in systems engineering is that complex systems are built by assembling components. Components have different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system. A central problem is the meaningful composition of heterogeneous components to ensure their correct interoperation. A fundamental source of heterogeneity is the composition of subsystems with different execution and interaction semantics. At one extreme of the semantic spectrum are fully synchronized components which proceed in a lockstep with a global clock and interact in atomic transactions. At the other extreme are completely asynchronous components, which proceed at independent speeds and interact non-atomically. Between the two extremes a variety of intermediate models can be defined (e.g. globally-asynchronous locally-synchronous models).

In this work, we study the combination of synchronous and asynchronous systems. To achieve this, we rely on BIP (Behavior-Interaction-Priority), a general component-based framework encompassing rigorous design. We define an extension of BIP, called Synchronous BIP, dedicated to model synchronous data-flow systems. Steps are described by acyclic Petri nets equipped with data and priorities. Petri nets are used to model concurrent flow of computation. Priorities are instrumental for enforcing run-to-completion in the execution of a step. We study a class of *well-triggered* synchronous systems which are by construction deadlock-free and their computation within a step is confluent. For this class, the behavior of components is modeled by *modal flow graphs*. These are acyclic graphs representing three different types of dependency between two events p and q : strong dependency (p must follow q), weak dependency (p may follow q), conditional dependency (if both p and q occur then p must follow q).

We propose translation of LUSTRE and discrete-time MATLAB/Simulink into well-triggered synchronous systems. The translations are modular and exhibit data-flow connections between components and their synchronization by using clocks. This allows for integration of synchronous models within heterogeneous BIP designs. Moreover, they enable the application of validation and automatic implementation techniques already available for BIP. Both translations are currently implemented and experimental results are provided.

For Synchronous BIP models we achieve efficient code generation. We provide two methods, sequential implementation and distributed implementation. The sequential implementation produces endless single loop code. The distributed implementation transforms modal flow graphs to a particular class of Petri nets, that can be mapped to Kahn Process Networks.

Finally, we study the theory of latency-insensitive design (LID) which deals with the problem of interconnection latencies within synchronous systems. Based on the LID design, synchronous systems can be “desynchronized” as networks of synchronous processes that might run with increased frequency. We propose a model for LID design in Synchronous BIP by representing specific LID interconnect mechanisms as synchronous BIP components.

Resumé

Une idée centrale en ingénierie des systèmes est de construire les systèmes complexes par assemblage de composants. Chaque composant a ses propres caractéristiques, suivant différents points de vue, chacun mettant en évidence différentes dimensions d'un système. Un problème central est de définir le sens la composition de composants hétérogènes afin d'assurer leur interopérabilité correcte. Une source fondamentale d'hétérogénéité est la composition de sous-systèmes qui ont des différentes sémantiques d'exécution et d'interaction. À un extrême du spectre sémantique on trouve des composants parfaitement synchronisés par une horloge globale, qui interagissent par transactions atomiques. À l'autre extrême, on a des composants complètement asynchrones, qui s'exécutent à des vitesses indépendantes et interagissent non-atomiquement. Entre ces deux extrêmes, il existe une variété de modèles intermédiaires (par exemple, les modèles globalement asynchrones et localement synchrones).

Dans ce travail, on étudie la combinaison des systèmes synchrones et asynchrones. A ce fin, on utilise BIP (Behavior-Interaction-Priority), un cadre général à base de composants permettant la conception rigoureuse de systèmes. On définit une extension de BIP, appelée BIP synchrone, destiné à modéliser les systèmes flot de données synchrones. Les pas d'exécution sont décrites par des réseaux de Petri acycliquement munis de données et des priorités. Ces réseaux de Petri sont utilisés pour modéliser des flux concurrents de calcul. Les priorités permettent d'assurer la terminaison de chaque pas d'exécution. Nous étudions une classe des systèmes synchrones "well-triggered" qui sont sans blocage par construction et le calcul de chaque pas est confluent. Dans cette classe, le comportement des composants est modélisé par des 'graphes de flux modaux'. Ce sont des graphes acycliques représentant trois différents types de dépendances entre deux événements p et q : forte dépendance (p doit suivre q), dépendance faible (p peut suivre q) et dépendance conditionnelle (si p et q se produisent alors p doit suivre q).

On propose une transformation de modèles LUSTRE et MATLAB/Simulink discret à temps discret vers des systèmes synchrones "well-triggered". Ces transformations sont modulaires et explicitent les connexions entre composants sous forme de flux de données ainsi que leur synchronisation en utilisant des horloges. Cela permet d'intégrer des modèles synchrones dans les modèles BIP hétérogènes. On peut ensuite utiliser la validation et l'implantation automatique déjà disponible pour BIP. Ces deux traductions sont actuellement implémentées et des résultats expérimentaux sont fournis.

Pour les modèles BIP synchrones nous parvenons à générer du code efficace. Nous proposons deux méthodes : une implémentation séquentielle et une implémentation distribués. L'implémentation séquentielle consiste en une boucle infinie. L'implémentation distribuée transforme les graphes de flux modaux vers une classe particulière de réseaux de Petri, que l'on peut transformer en réseaux de processus de Kahn.

Enfin, on étudie la théorie de la conception de modèles insensibles à la latence (latency-insensitive design, LID) qui traite le problème de latence des interconnexions dans les systèmes synchrones. En utilisant la conception LID, les systèmes synchrones peuvent être «désynchronisés» en des réseaux de processus synchrones qui peuvent fonctionner à plus haute fréquence. Nous

proposons un modèle permettant de construire des modèles insensibles à la latence en BIP synchrone, en représentant les mécanismes spécifiques d'interconnexion par des composants BIP synchrone.

Chapter 1

Introduction

Motivation of this thesis

The last decades computer technology has become ubiquitous. Computer systems are used for a wide range of tasks, embedded in many forms. Examples can be found in consumer electronics such as mobile phones, house electrical appliances and in industries like avionics, aerospace and nuclear plants. We call these systems *embedded systems*. Embedded systems constitute a domain where there is a special need for rigorous design methods. Such methods require formal frameworks to model the system at different design stages, from specification to implementation, and formal techniques to assess its correctness and performance.

In this context, *the component-based design* has been established as an important paradigm for the development of embedded systems. The main principle is that complex systems can be obtained by assembling components (building blocks) [67]. Components are systems characterized by their interface, an abstraction that is adequate for composition and reuse. Composition is used to build complex components by “gluing” together simpler ones. “Gluing” can be seen as an operation that takes in components and their integration constraints. From these, it provides the description of a new, more complex component.

Embedded systems are often built from heterogeneous components [9]. A common source of heterogeneity concerns on different execution paradigms. On one hand, the *synchronous execution paradigm*, widely accepted for the design of hardware components. It considers systems that are designed as the composition of parallel components which are strongly synchronized. These components proceed in lock-step with a global clock and interact in atomic transactions. In each execution step, all the system components contribute by executing some quantum of computation. The synchronous execution paradigm, therefore, has a built in strong assumption of fairness: in each step all components can move forward. On the other hand, the *asynchronous execution paradigm*, used for the design of software components. It considers systems, designed from sequential components which are completely asynchronous. These components proceed at independent speeds and interact nonatomically. This execution model is adopted in most distributed systems description languages such as UML, and in multi threaded programming languages such as ADA and Java. The lack of built in mechanisms for sharing computation between components can be compensated through scheduling mechanisms, e.g., priorities.

However, for general applications, an adequate mix of synchronous and asynchronous computation is demanded e.g. GALS models. Many recent microprocessor designs address the GALS design challenge. Modern system-on-a-chip (SoC) products migrate from fully synchronous design to GALS designs. In GALS designs, each core is a synchronous block of logic while communication between cores is asynchronous [47]. The core interface logic is usually considered to be a wrapper around the synchronous functional logic. Cores which are critical to system performance run at higher frequencies, while less critical cores run at lower frequencies to conserve

power. The paradigm shift to GALS is being driven by the impracticality of fully synchronous design for large SoCs.

Presently, there is a lack of formalisms encompassing both synchronous and asynchronous execution. Encompassing heterogeneity of execution semantics is the vision that motivates this thesis. This requires in principle, the use of common semantic model encompassing both the synchronous and the asynchronous formalisms.

Context of this thesis

Two are the main domains that constitute the context of this thesis, the rigorous component-based design of embedded systems and the synchronous execution paradigm.

To encompass heterogeneity of execution we need to rely on a component-based framework which provides rigorous semantics. *BIP (Behavior, Interaction, Priority)* is such a formalism for modeling heterogeneous component-based systems [12], developed in Verimag. It allows the description of systems as the composition of generic atomic components characterized by their behavior and their interfaces. It supports a system construction methodology based on the use of two families of composition operators: interactions and priorities. Interactions are used to specify multiparty synchronization between components as the combination of two protocols: rendezvous (strong symmetric synchronization) and broadcast (weak asymmetric synchronizations). Priorities between interactions are used to restrict non determinism inherent to parallel systems. They are particularly useful to model scheduling policies.

In contrast to existing formal frameworks, BIP is expressive enough to directly model any coordination mechanism between components [23]. It has been successfully used to model complex systems including mixed hardware/software systems and complex software applications. BIP can be used as a unifying semantic model for structural representation of different, domain specific languages and programming models, as illustrated in figure 1.1.

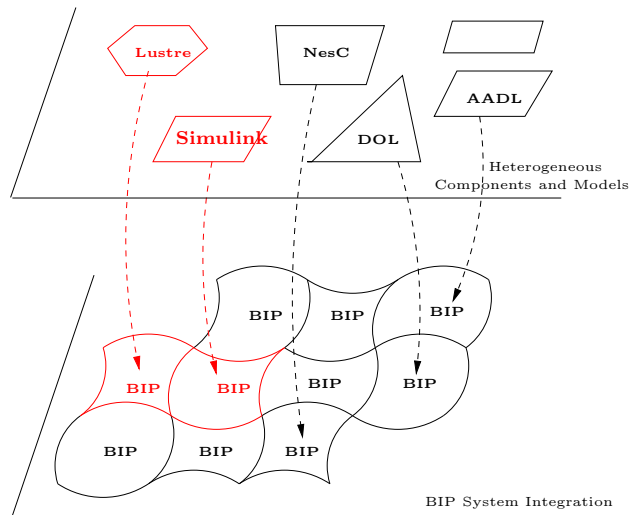


Figure 1.1: The Language Factory of BIP

A general method has been established for generating BIP models from languages with well-defined operational semantics. This method involves the following three steps. First, the source language is translated/transformed into BIP components. The translation focuses on the definition of adequate interfaces. It encapsulates and reuses data structures and behavior of the original components. Second, it translates coordination mechanisms between components of

the source language into connectors and priorities in the BIP model. Third, it generates a BIP component, modeling the operational semantics of the source language. This component plays the role of an engine that coordinates the overall execution. It is actually needed only if specific execution constraints, (that are not directly captured by coordination through connectors and priorities) need to be enforced. There have been developed BIP model generators for several programming models used by embedded system developers including the *Architecture Analysis and Design Language* AADL[32], NesC/TinyOS[14], the *Distributed Operation Layer* DOL [60], the programming model *GeNoM* [13], etc. The generated models preserve the structure and their size is linear with respect to the size of the initial programs. Furthermore, they are easy to understand by developers in source languages.

Synchronous programming is a design method for modeling, specifying, validating and implementing safety critical applications [18]. The synchronous paradigm provides ideal primitives which allow a program to be considered as instantaneously reacting to external events [42].

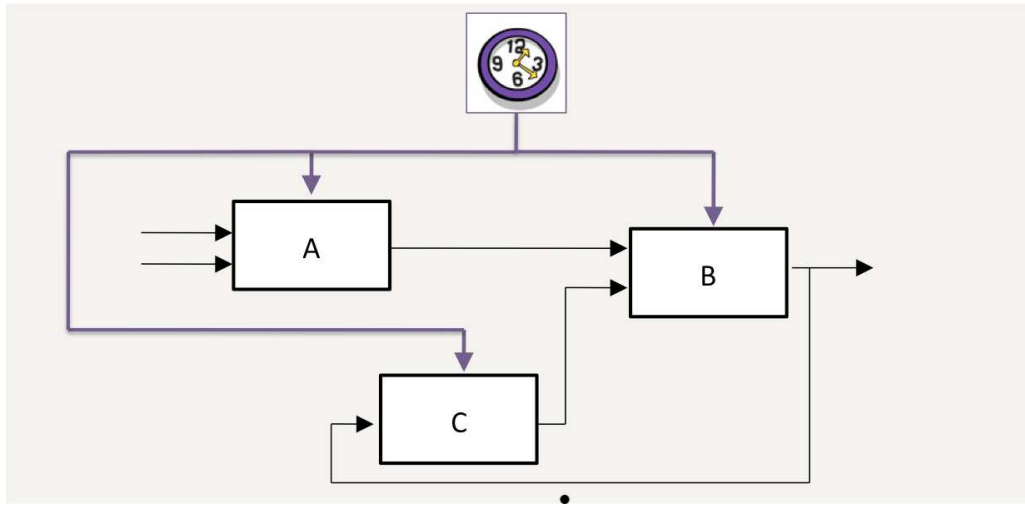


Figure 1.2: A Synchronous System

Synchronous systems, as shown in Figure 1.2, consist of a network of parallel blocks/operators (A,B,C,...) the execution of which is triggered by a global clock. This clock produces successive “clock ticks” which divide the computation of a synchronous program in *execution instants* (synchronous steps). Inside each instant (step), input signals occur, internal computations take place and data is propagated to the outputs. Computations are performed instantaneously and take place as a reaction to external events. A program reacts fast enough to receive and to proceed with all external events in suitable order. In addition, the communications between different processes are performed via instantaneous broadcasting which are considered to “take no time”.

Synchronous processes are composed in parallel. Parallel composition helps in structuring the model, without introducing non-determinism. The *determinism* of the model is an invaluable advantage for its understanding, validation and the verification. *Concurrency* is another important property for synchronous programming. Programs can be decomposed into subunits and be executed in parallel. During each step, each subunit reacts instantaneously to triggering events and communicates with other subunits instantaneously. This decomposition leads to readable, maintainable and reusable components.

Synchronous programming is based on mathematical principles that makes possible handling the compilation, verifying the programs in a formal way and proving logical correctness [16] defined with respect to input/output specification.

Contributions of this thesis

This work aims to extend the BIP component-based framework by building a framework dedicated to modeling synchronous data-flow systems. The benefits of this extension are two-folded. First, it presents a general approach for modeling synchronous component-based systems. Synchronous formalisms such as the LUSTRE language and the Simulink framework can be translated to the extension of BIP and be represented as Synchronous components. The definition of synchronous components as an extension of the BIP framework allows their combination with other asynchronous languages that can be translated into BIP (see Figure 1.1). Second, it opens the way for studying combination of synchronous and asynchronous systems. It allows integration of synchronous systems theory in all encompassing component framework [23] without losing advantages such as correctness-by-construction and efficient code generation. This allows modeling mixed synchronous/asynchronous systems without artefacts.

The contributions that this thesis brings are the following:

- We present the *Synchronous BIP framework*, an extension of BIP for modeling synchronous data-flow systems. We define a notion of *synchronous BIP component* which differs from general components in that its behavior is described by a step. The behavior of a component in a *step* is described by a safe extended priority Petri net. We define composition of synchronous components as a partial internal operation parametrized by a set of *interactions*. We define the class of *modal flow components* where priority Petri nets are replaced by *modal flow graphs*. These graphs correspond to a subclass of priority Petri nets for which deadlock-freedom and confluence can be decided at low cost. Modal flow graphs are structures expressing dependency relations between events.
- We translate the LUSTRE language and the MATLAB/Simulink framework into Synchronous BIP. Both translations are modular and make explicit all the interactions needed to perform a synchronous computation in an inherently parallel (component-based) system. Moreover, they exhibit data-flow connections between components and their synchronization by using clocks. This allows for integration of synchronous models within heterogeneous BIP designs. In addition, they enable the application of validation, verification and automatic implementation techniques already available for BIP. Both translations are currently implemented and experimental results are provided.
- We provide a method for generating sequential code. This method produces endless single loop C code. We provide tool that implement this method and we report results for several examples. We compare performances with C code generated from the LUSTRE and MATLAB native code generators.
- We provide two methods for generating distributed code, the “direct” method and the “cluster-oriented” method. Generation of code is done in two steps. First, they transform modal flow graphs to Petri nets. Second, they map the produced Petri nets to Kahn process networks.
- We propose a desynchronization of Synchronous BIP components based on the theory of Latency-Insensitive Design (LID). This theory deals with the problem of interconnection latencies within synchronous systems. Based on the LID design, synchronous systems can be “desynchronized” as networks of synchronous processes that might run with increased frequency. We propose a model for LID design in Synchronous BIP by representing specific LID interconnection mechanisms as synchronous BIP components.

Organization of this document

The rest of this document is structured as follows. Chapter 2 describes the BIP component-based framework which is the foundation of this work. Chapter 3 gives an introduction to synchronous languages and provides the basics of LUSTRE language and MATLAB/Simulink. Chapter 4 describes the Synchronous BIP framework. The translations of LUSTRE and Simulink to Synchronous BIP are provided in Chapter 5. Chapter 6 describes the sequential and distributed methods for code generation from Synchronous BIP models. Chapter 7 proposes a method for Latency-Insensitive Design in Synchronous BIP. Chapter 8 draws the conclusions of this work and possible future directions.

Chapter 2

The BIP Framework

Component-based design is a paradigm that wants complex systems to be obtained by assembling components. Components are characterized by abstractions that ignore implementation details and describe properties relevant to their composition. Composition is used to build complex components from simpler ones.

In this chapter, we present the *BIP* [12, 40] (Behavior, Interaction, Priority) component-based design framework encompassing heterogeneous composition. A BIP component consists of the superposition of three layers: behavior, interaction and priority, as shown in Figure 2.1.

BIP allows the description of systems as the composition of generic atomic components characterized by their behavior and their interfaces. It supports a system construction methodology based on the use of two families of composition operators: *interactions* and *priorities*. Interactions are used to specify multiparty synchronization between components. Priorities between interactions are used to restrict non determinism between interactions simultaneously enabled. They are particularly useful to model scheduling policies.

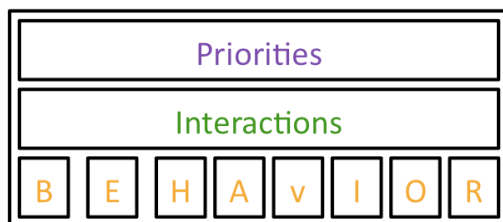


Figure 2.1: Structure of a BIP model

Complex components are obtained by “gluing” together simpler components [67]. Composition of components provides the description of a new component based on the integration characteristics and constraints of a set of atomic components, by composing their corresponding layers separately.

This chapter is structured as follows. The abstract model of BIP is described in section 2.1. In this model, the behavior of atomic components is described as a labeled transition system. Section 2.2 describes the concrete model of BIP. In this model, atomic components are described as Petri nets extended with data. Interactions provide data transfer between components. We define the operational semantics for all three layers (behavior, interaction, priorities) giving additional information on the functionality of atomic components and on valuation of data in atomic components and interactions. We illustrate the concrete model of BIP using the Precision Time Protocol example [37].

The basic constructs of the BIP language are described in section 2.3. Section 2.4 describes the BIP toolset and the code generation for BIP models. Finally, section 2.5 draws some

conclusions.

2.1 Abstract Model of BIP

2.1.1 Behavior

An *atomic component* is the most basic BIP component which represents behavior and has empty interaction and priority layers. A formal definition for the behavior of an atomic BIP component is given below:

Definition 1 (Behavior) *A behavior B of an atomic component is a labeled transition system (LTS) represented by a triple (Q, P, \rightarrow) where:*

- Q is a set of control states,
- P is a set of ports,
- $\rightarrow \subseteq Q \times P \times Q$ is a set of transitions.

For a pair of states $q, q' \in Q$ and a port $p \in P$, we write $q \xrightarrow{p} q'$, iff $(q, p, q') \in \rightarrow$ and we say that p is enabled at q . If such q' does not exist, we write $q \not\xrightarrow{p}$ and we say that p is disabled at q .

A set of atomic components can be combined together by using a special “glue”. A glue \mathcal{GL} is a separate layer, composing the underlying layer of behaviors. It is a set of operators mapping tuples of behavior into behavior. The BIP component framework uses two models of glue for composition of behavior, the *interaction* model and the *priority* model.

2.1.2 Abstract Model of Interactions

Let $\{B_i = (Q_i, P_i, \rightarrow_i)\}_{i=1}^n$ be a set of atomic components and $P = \bigcup_{i=1}^n P_i$ be the set of all ports.

We consider that for components, the respective sets of ports and the sets of states are pairwise disjoint i.e., for all $i \neq j$, we have $P_i \cap P_j = \emptyset$ and $Q_i \cap Q_j = \emptyset$ respectively.

The following definition describes an interaction and a connector.

Definition 2 (Interaction, Connector) *An interaction a is a non-empty subset of ports i.e. $a \subseteq P$ such that $\forall i \in \overline{1, n}, |a \cap P_i| \leq 1$. A connector γ is defined as a set of interactions that is $\gamma \subseteq 2^P$.*

Interactions a of γ can be enabled or disabled. An interaction a is enabled iff $(\forall p \in P, \text{ with } p \in a, p \text{ is enabled})$. That is, an interaction is enabled if each port that is involved in this interaction, is enabled. An interaction a is disabled iff $(\exists p \in P, \text{ with } p \in P \text{ such that } p \text{ is disabled})$. That is, an interaction is disabled if there exists at least a port, involved in this interaction, that is disabled.

Connectors are described using algebraic formalisms as shown in [23]. They are modelled as terms of the algebra of connectors $\mathcal{AC}(P)$, generated from a set of ports P by using special operators. The semantics of $\mathcal{AC}(P)$ associates with a connector the set of its interactions.

2.1.3 Abstract Model of Priorities

In a behavior, more than one interactions can be enabled at the same time, introducing a degree of non-determinism. This can be restricted with priorities by filtering the possible interactions based on the current global state of the system. The formal definition for a priority is given below.

Definition 3 (Priority) *A priority is a relation $\prec \subseteq \gamma \times Q \times \gamma$, where γ is the set of interactions, and Q is the global set of states.*

For $a \in \gamma$, $q \in Q$ and $a' \in \gamma$, the priority $(a, q, a') \in \prec$ is denoted as $a \prec_q a'$. This relation says that interaction a has less priority than interaction a' at state q . Furthermore, we require that for all $q \in Q$, \prec_q is a strict partial order on γ .

2.1.4 Composition of Atomic Components

The interaction model γ is a set of interactions. The priority model is a set of priorities π . The glue \mathcal{GL} is composed of the two previous models γ and π and defined as $\mathcal{GL} = \pi\gamma$. For a set of components $\{B_i = (Q_i, P_i, \rightarrow_i)\}_{i=1}^n$, an interaction model γ and a priority model π , the compound component is obtained by application of the glue $\pi\gamma$, i.e. $\pi\gamma(\{B_i\}_{i=1}^n)$. An interaction is enabled in $\pi\gamma(\{B_i\}_{i=1}^n)$ only if it is enabled in γ and maximal according to π among the enabled interactions in $\{B_i\}_{i=1}^n$.

The following definitions provide the operational semantics for the composition of a system of behavior with respect to an interaction model and restricted from the priority model respectively.

Definition 4 (Composition for Interaction Model) *The composition of a set of components $\{B_i = (Q_i, P_i, \rightarrow_i)\}_{i=1}^n$ is a transition system represented by the triple $(Q, \gamma, \rightarrow_\gamma)$, where:*

- $Q = \otimes_{i=1}^n Q_i$,
- γ is the set of interactions $\gamma \subseteq 2^P$ where $P = \cup_{i=1}^n P_i$ and
- \rightarrow_γ is the least set of transitions defined by the rule:

$$\frac{a = \{p_i\}_{i \in I} \in \gamma, \quad I \subseteq \overline{1, n} \quad (\forall i \in I : \{q_i \xrightarrow{p_i} q'_i\}), \quad (\forall i \notin I : q'_i = q_i),}{(q_1, \dots, q_n) \xrightarrow{a}_\gamma (q'_1, \dots, q'_n)}$$

The rule says that the obtained behavior, that we will note as $\gamma(B_1, \dots, B_n)$, can execute a transition $a \in \gamma$, iff for each $i \in I$, port p_i is enabled in B_i .

Definition 5 (Composition restricted from the Priority Model) *Given a behavior $B = (Q, \gamma, \rightarrow_\gamma)$, its restriction by the priority model π is the behavior $B' = (Q, \gamma, \rightarrow_\pi)$ defined by the rule*

$$\frac{a \in \gamma \quad q \xrightarrow{a}_\gamma q', \quad (\forall a' \in \gamma : a \prec_q a') \Rightarrow q \not\xrightarrow{a'}_\gamma}{q \xrightarrow{a}_\pi q'}$$

The rule says that the obtained behavior γ can execute a transition $a \in \gamma$ iff each transition $a' \in \gamma$, with higher priority than a in state q , is disabled.

2.2 Concrete Model of BIP

In this section we give formal definitions for the concrete model of BIP. We illustrate the use of BIP by modeling a concrete example, the Precision Time Protocol (PTP) [37].

Running Example: The Precision Time Protocol

PTP is a high precision time protocol for synchronizing multiple clocks. The protocol defines synchronization messages used between a Master and one or many Slave clocks. The Master clock is the provider of time and a Slave clock synchronizes to the Master. The communication from the Master to a Slave and from a Slave to the Master is done through specific messages. Precise timestamps are captured at the Master and Slave clock and are used to determine the latency of the Slave clock. As shown in Figure 2.2 these timestamps are referred to as t_1, t_2, t_3, t_4 . There is a sync message transmitted periodically from the Master clock which contains the time t_1 of the Master clock. The sync message is received by the Slave clock at time t_2 . The timestamp t_1 is transmitted from the Master clock to the Slave clock via the message followUp. A request message is transmitted from the Slave clock at timestamp t_3 . The timestamp t_4 represents the time that the request message was received at the Master clock. The timestamp t_4 is transmitted from the Master clock to the Slave clock via the message reply. The offset o is calculated as $((t_2 - t_1) - (t_4 - t_3))/2$ and it is utilized by the Slave clock to adjust to the time to agree with the Master clock. The protocol assures the communication delays between the Master and the Slave to be equal.

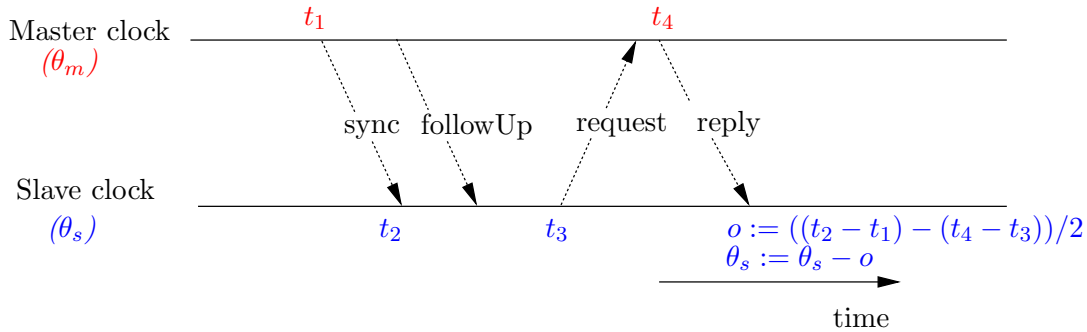


Figure 2.2: PTP behavior

2.2.1 Modeling BIP Atomic Components

An *atomic BIP component* represents behavior and has empty interactions and priority layers. The formal definition of a BIP atomic component is given below.

Definition 6 (Atomic BIP component:syntax) An atomic component B is a tuple (X, P, N) where:

- X is a set of data variables
- P is a set of ports p , each one labelled with a subset of variables $X_p \subseteq X$, the ones exported on interactions through p .
- $N = (L, T, F, L_0)$ is an 1-safe Petri net:

- L is a finite set of places
- T is a finite set of transitions τ labelled by $(p_\tau, gu_\tau, f_\tau)$ where:
 - * p_τ is the port triggered by the transition τ ,
 - * gu_τ is the guard of τ and it is a predicate on X .
 - * f_τ is the update function associated with the transition τ . $f_\tau = (f_\tau^{(x)})_{x \in X}$, that is, for every $x \in X$, it provides an arbitrary expression on X defining the next (updated) value for x . We concretely represent f_τ as sequential programs operating on data X .
- $F \subseteq L \times T \cup T \times L$ is the token flow relation,
- $L_0 \subseteq L$ is the set of initial places.

Let us remark that, within an atomic component, variables attached to ports can overlap. That is, for ports p_i, p_j of an atomic component with $i \neq j$ and for their associated variables X_{p_i} and X_{p_j} respectively, it holds $X_{p_i} \cap X_{p_j} \neq \emptyset$.

Graphically, an atomic component is represented as a box. The behavior of an atomic component is represented as a Petri net. Each transition is labelled with a port, a guard and an update function. Ports and variables associated to ports are represented as boxes and shown on the border of the atomic component.

Example 1 Figure 2.3(left) shows the Master_clock BIP atomic component that corresponds to the Master clock for the PTP model. It has five ports tick, sync, followUp, reply and request and variables x, t_1, t_4 and θ_m where t_1 and t_4 are associated with the ports followUp and reply respectively. The set of places is $\{q_1, q_2, q_3, q_4\}$.

Initially, the tick transition can be executed. This transition is associated with the update function $f_{tick} = (f_{tick}^x, f_{tick}^{t_1}, f_{tick}^{t_4}, f_{tick}^{\theta_m})$ with $f_{tick}^x = x + 1$, $f_{tick}^{\theta_m} = \theta_m + 1$. We represent concretely this function as $f_{tick} : x = x + 1; \theta_m = \theta_m + 1$. That is, when the tick transition is executed, it increments by one the local variable x and the Master_clock θ_m . Whenever x reaches the value P , the sync transition is executed, the variable x is reset to zero and the timestamps t_1 records the time of the Master_clock when the transition sync took place. The execution of sync is followed by the execution of followUp which emits the timestamps t_1 . Two executions can follow; either tick is executed increasing the values of θ_m and of x by one, or the request transition is executed recording in t_4 the actual time of the Master_clock. The transition request is followed by the execution of reply and the emission of the t_4 timestamp.

Example 2 Figures 2.3 (right) and 2.4 show the BIP atomic components for the Slave_clock and the Master_to_Slave and Slave_to_Master respectively.

The Slave_clock component is dual to the Master_clock. Each time the tick transition is executed the value of the variable θ_s , that represents the value of the Slave_clock, is incremented by one. When sync is executed, the value of the Slave_clock is recorded at the timestamp t_2 . When the transitions followUp and reply are executed, they receive the t_1 and t_4 variables respectively. Moreover, when reply is executed, the offset between the Slave clock and the Master clock is computed and the Slave clock is adjusted.

The Master_to_Slave channel and the Slave_to_Master channel are abstraction of the communication network between the Master_clock and the Slave_clock.

We consider that the Master_to_Slave component (Figure 2.4 (left)) executes initially the inSync transition and resets the value of y to zero. Transition inrc₁ increases the variable y by one till the moment that the transition inFollowUp is executed. At that moment, the value of z is reset to zero. Similarly to incr₁, when incr₂ is executed, it increases the value of z by one.

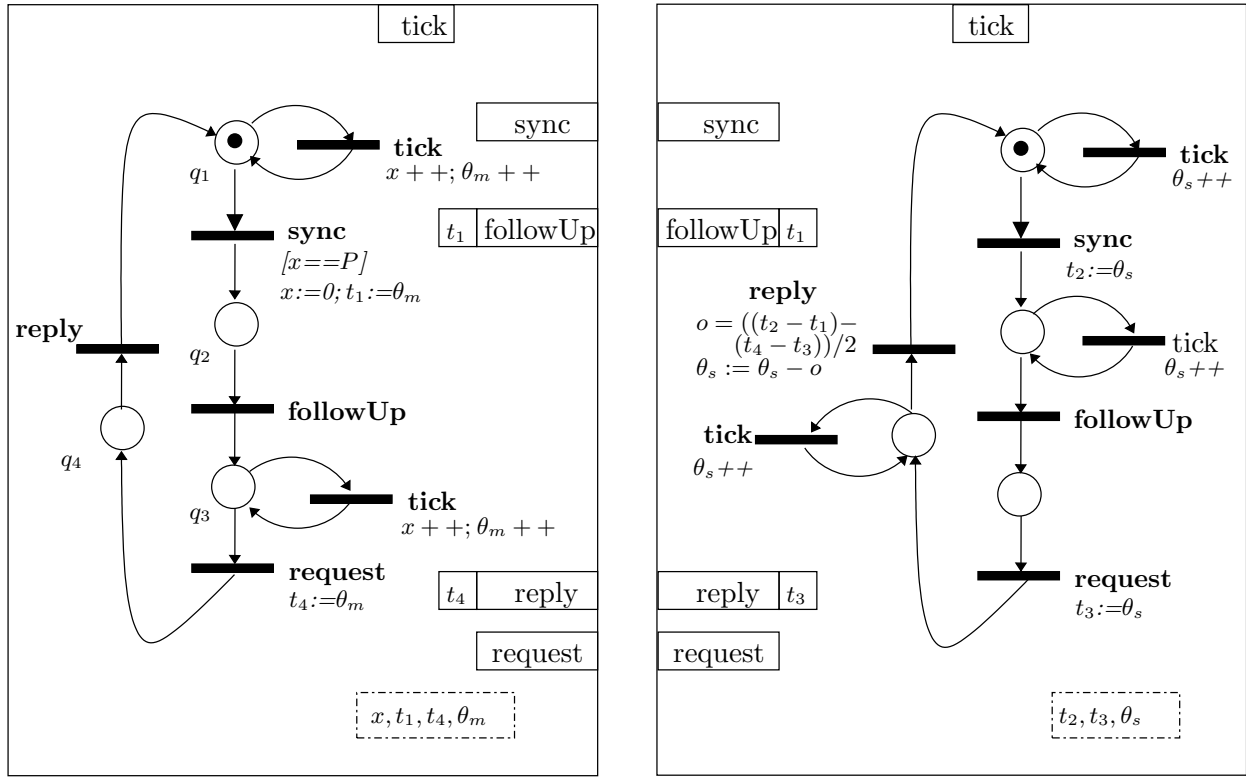


Figure 2.3: The Master_clock (left) and the Slave_clock (right) BIP atomic components

The transitions *outSync* and consecutively, *outFollowUp*, are executed if the “delay” values y and z satisfy the arbitrary delays bounded in the intervals (L, U) . The execution continues with the transition *inReply* which resets n to zero, the transition *incr₃* which increases the value of n by one each time it is triggered and finally, with the transition *outReply* which is triggered if the “delay” value n satisfies the arbitrary delays bounded in the intervals (L, U) .

The *Slave_to_Master* channel component (Figure 2.4 (right)) executes initially the transition *outRequest* resetting the value of m to zero. Each time the transition *incr₃* is executed, the value of n is increased by one till the moment the transition *inRequest* is executed. The execution of this transition is restricted from the arbitrary delays bounded in the intervals (L, U) .

2.2.2 Semantics of Atomic Components

In order to define the operational semantics for atomic BIP components, let us first introduce some notations.

We assume 1-safe Petri nets, that is Petri nets with at most one token per place. Given a Petri net $N = (L, T, F, L_0, L_f)$ the set of 1-safe markings \mathcal{M} is the set of functions $m : L \rightarrow \mathbb{N}$. Given two markings m_1, m_2 , inclusion $m_1 \leq m_2$ holds iff for all $l \in L$, $m_1(l) \leq m_2(l)$. Also, addition $m_1 + m_2$ is the marking m_{12} such that, for all $l \in L$, $m_{12}(l) = m_1(l) + m_2(l)$. Given a set of places $K \subseteq L$, we define its characteristic marking m_K by $m_K(l) = 1$ for all $l \in K$ and $m_K(l) = 0$ for all $l \in L \setminus K$. Moreover, when no confusion is possible from the context, we will simply use K to denote its characteristic marking m_K . Finally, for a given transition τ , its pre-set $\bullet\tau$ (resp. post-set $\tau\bullet$) is the set of places flowing to (resp. from) that transition $\bullet\tau = \{l \mid (l, \tau) \in F\}$ (resp. $\tau\bullet = \{l \mid (\tau, l) \in F\}$).

We assume a universal data domain \mathcal{D} . Given a set of data variables X , we define valuations

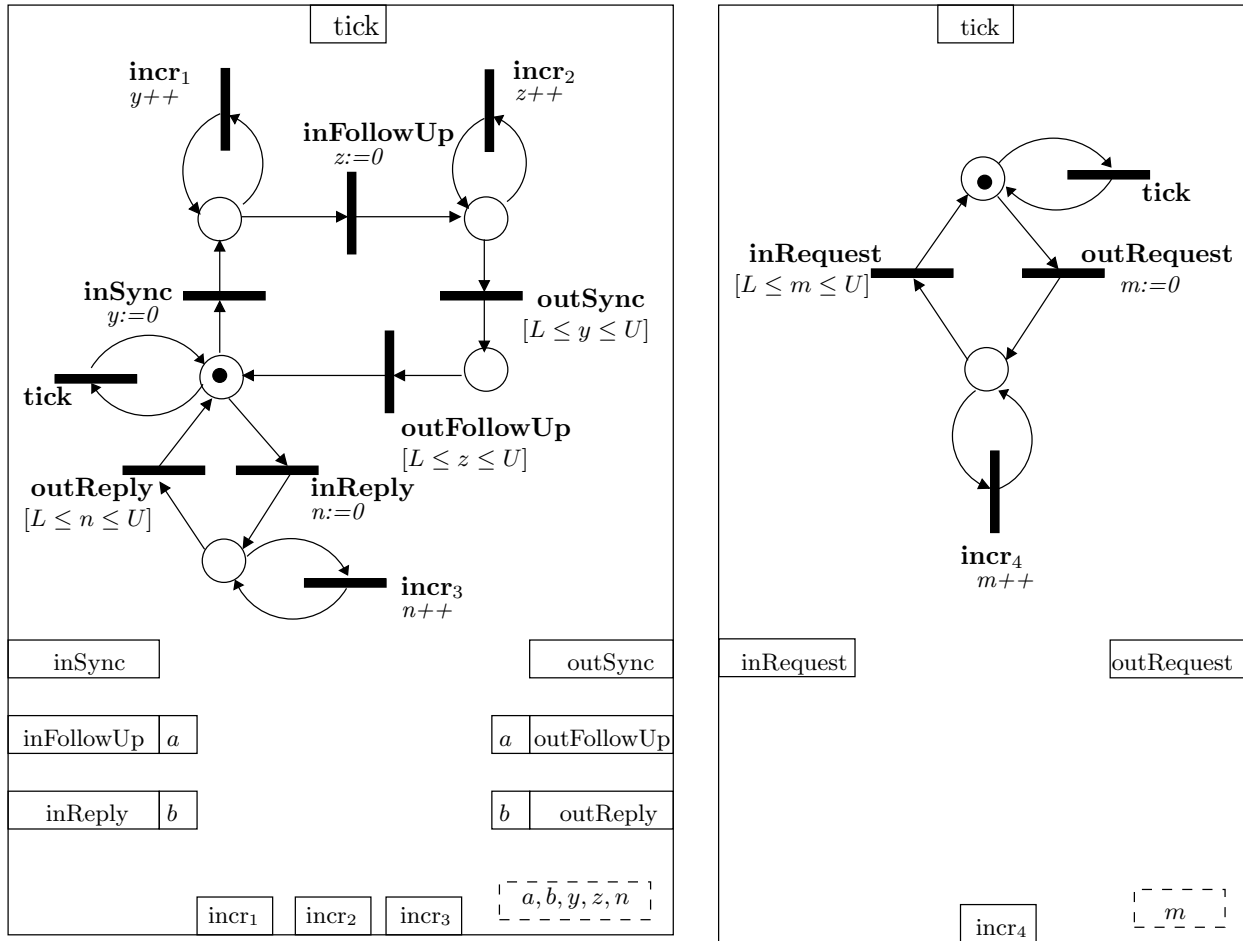


Figure 2.4: The Master_to_Slave (left) and the Slave_to_Master (right) BIP atomic components

for X as functions $u : X \rightarrow \mathcal{D}$. The set of valuations is noted as \mathcal{D}^X .

We tacitly extend valuations to expressions defined on X . That is, for any expression e on X and u an valuation for X , we denote by $e(u)$ the value of e for the valuation u .

Given two valuations $u : X \rightarrow \mathcal{D}$ and $v : X' \rightarrow \mathcal{D}$, we define the sequential application $u \oplus v : X \cup X' \rightarrow \mathcal{D}$ as a valuation defined by:

$$(u \oplus v)(x) = \begin{cases} v(x) & : \text{if } x \in X' \\ u(x) & : \text{if } x \in X \setminus X' \end{cases}$$

Finally, given a valuation $u : X \rightarrow \mathcal{D}$ and $X' \subseteq X$, we denote by $u|_{X'}$ the restriction of u to variables in X' .

Definition 7 (Atomic BIP component: semantics) The operational semantics of an atomic BIP component $B = (X, P, N)$ with $N = (L, T, F, L_0)$ is defined as the labelled transition system $S = (Q, \Sigma, \rightarrow_B)$ where

- $Q = \mathcal{M} \times \mathcal{D}^X$ is the set of states where
 - $\mathcal{M} = \{m : L \rightarrow \mathbb{N}\}$ is the set of 1-safe markings
 - $\mathcal{D}^X = \{u : X \rightarrow \mathcal{D}\}$ is the set of valuation of data X on the domain \mathcal{D}
- $\Sigma = \{(p, v, v') \mid p \in P, v \in \mathcal{D}^{X_p}, v' \in \mathcal{D}^{X_p}\}$ is the set of labels. A label (p, v, v') marks instantaneous data change through the port p . The current valuation v is sent and a new valuation v' is received for the set of variables X_p . We note a label (p, v, v') as $p(v/v')$.
- $\rightarrow_B \subseteq Q \times \Sigma \times Q$ is the set of transitions defined by the following rule:

	control	data		
$\tau \in T$	$m \in \mathcal{M}, m' \in \mathcal{M}$	$u \in \mathcal{D}^X, u' \in \mathcal{D}^X$		
labeled by		$v \in \mathcal{D}^{X_{p_\tau}}, v' \in \mathcal{D}^{X_{p_\tau}}$		
$(p_\tau, gu_\tau, f_\tau)$	$\bullet\tau \leq m$	$gu_\tau(u) = \text{true}$	(read u)	guard
and $X_{p_\tau} \subseteq X$		$v = u _{X_{p_\tau}}$	(read v)	communication
the set of		$v' : \text{arbitrary}$	(write v')	
variables for p_τ	$m' = m - \bullet\tau + \tau\bullet$	$u' = f_\tau(u \oplus v')$	(write u')	action

$$(m, u) \xrightarrow{p_\tau(v/v')} (m', u')$$

This rule corresponds to the firing of a transition τ labeled by the port p_τ , the guard gu_τ and the update function f_τ of a BIP component. A transition can be executed depending on the marking m and the valuation of its guard gu_τ . The guard is evaluated on the current valuation u of the set of data of the component. The execution of a transition includes the following micro-steps:

- An instantaneous data change through the port p is performed: the current valuation v is sent and a new valuation v' is received for variables in X_{p_τ} ,
- The next state valuation u' as defined by f_τ , is computed using the new valuation v' together with the current valuation u ,
- The marking is updated to m' according to the net flow.

Composition of atomic components allows to build a system as a set of atomic components that interact by respecting constraints of an interaction model and a priority model.

2.2.3 Concrete Model of Interactions

Definition 8 (Interaction) An interaction a is a triple (P, G, F) where

- P is a set of ports, the support set of the interaction,
- G is the interaction guard, that is a boolean predicate defined on variables $X = \cup_{p \in P} X_p$ exported through ports belonging to the interaction.
- F defines the data transfer function associated with the transition $F = (F^{(x)})$, for every $x \in X$ where $X = \cup_{p \in P} X_p$. $F^{(x)}$ is an arbitrary expression on X defining the next updated value for x . We concretely represent F as sequential programs operating on data X .

We assume that connectors contain at most one port from each atomic component. In addition, we consider that connectors may be associated with a set of guarded commands, associated with feasible interactions. An interaction consists of one or more ports of the connector, a guard on the variables of the ports of the interaction and a function realizing data transfer between ports of the interactions.

Let $\{B_i = (X_i, P_i, N_i)\}_{i=1}^n$ be set of atomic components. We consider that the set of ports and the set of variables of different atomic components are disjoint.

The following definition describes a connector.

Definition 9 (Connector) A connector γ is a set of ports of atomic components B_i which can be involved in an interaction. It is defined as $\gamma = (P_\gamma, A_\gamma)$ where:

- P_γ is the support set of γ , that is, the set of ports that γ synchronizes
 - $\forall i \in \overline{1, n}, |P_\gamma \cap P_i| \leq 1$, that is, each connector γ uses at most one port from each component i .
- $A_\gamma \subseteq \mathcal{P}^{P_\gamma}$ is a set of interactions a each labeled by the triple (P_a, G_a, F_a) where:
 - P_a is the set of ports $\{p_i\}_{i \in I}, I \subseteq \overline{1, n}$ that take part in an interaction a ,
 - G_a is the guard of a , a predicate defined on variables $\bigcup_{p_i \in a} X_{p_i}$,
 - F_a is the data transfer function of a defined on variables $\bigcup_{p_i \in a} X_{p_i}$.

In BIP, we distinguish two models of synchronization on connectors:

- *Strong synchronization* or *rendezvous*, where the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ . We note $A_\gamma = P_\gamma$;
- *Weak synchronization* or *broadcast*, where all feasible interactions are those containing a particular port p_{trig} which initiates the broadcast. We note $A_\gamma = \{a \in \gamma \mid a \cap \{p_{trig}\} \neq \emptyset\}$ where $p_{trig} \in P_\gamma$ is the port that initiates the broadcast.

There is a graphical notation for interactions. In a *rendezvous* interaction all ports (known as *synchrons*) are denoted by bullets. In a *broadcast* interaction, the port that initiates the interaction, also called *trigger*, is denoted by a triangle and all the rest with bullets.

Example 3 Figure 2.5 shows the *Master_clock* and the *Master_to_Slave* BIP atomic components.

The ports of the *Master_clock* *sync*, *followUp* and *reply* are trigger ports and the ports *tick* and *request* are of type *synchron*. The *gtick* connector is a *rendezvous* synchronization that is, the only feasible interaction is $\{\text{Master_clock.tick Master_to_Slave.tick}\}$. The connectors

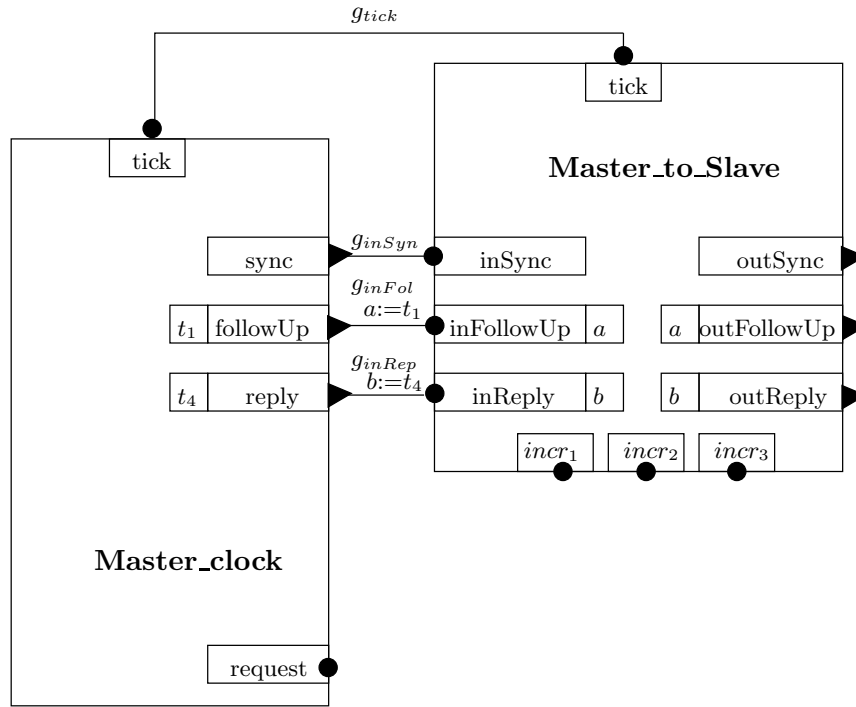


Figure 2.5: Synchronization between atomic components

g_{inSyn} , g_{inFol} and g_{inRep} are broadcast synchronizations. For example, the g_{inSyn} synchronization is initiated by `sync` and the feasible interactions are $\{Master_clock.sync, Master_clock.sync \rightarrow Master_to_Slave.inSync\}$. The g_{inFol} interaction is associated with a data transfer between the `Master_clock` and the `Master_to_Slave` component. It is specified by the action $a := t_1$ that copies the value t_1 of the `Master_clock` to the value a of the `Master_to_Slave` channel.

2.2.4 Concrete Model of Priorities

Definition 10 (Priority) A priority is a tuple (C, \prec) where C is a state predicate (boolean condition) characterizing the states where the priority applies and \prec gives the priority order on a set of interactions $A = \bigcup A_\gamma$

For $a_1 \in A$ and $a_2 \in A$, a priority rule is textually expressed as $C \rightarrow a_1 \prec a_2$. When the state predicate C is true and both interactions a_1 and a_2 specified in the priority are enabled, the higher priority interaction, i.e., a_2 is selected for execution.

Example 4 For the composition of Figure 2.5 there is a non deterministic choice between the two interactions g_{tick} and g_{inSyn} . This is due to the behavior of the `Master_clock` that creates an execution conflict between the `tick` transition and the `sync` transition when the guard of the latter is evaluated true. Non determinism is resolved by the priority $true \rightarrow g_{tick} \prec g_{inSyn}$, which selects the interaction g_{inSyn} by disabling g_{tick} .

2.2.5 Composition of Atomic Components

Definition 11 (Composition:semantics) Let $\{B_i = (X_i, P_i, N_i)\}_{i=1}^n$ be set of atomic components and $S_i = (Q_i, \Sigma_i, \rightarrow)$ be the labeled transition system of B_i as presented in Definition 7. For

a connector γ , the semantics of the composition $\gamma(B_1, \dots, B_n)$ is defined as the labeled transition system $(Q, \Sigma, \rightarrow_\gamma)$ where:

- $Q = \otimes_{i=1}^n Q_i$
- Σ the set of labels such that $\Sigma = \gamma$, where each label corresponds to an interaction
- $\rightarrow_\gamma \subseteq Q \times \Sigma \times Q$ defined by the rule:

Steps		Synchronization		
$a = \{p_i\}_{i \in I} \in \gamma$	$(\forall i \in I : q_i \xrightarrow{p_i(v_i/v'_i)} q'_i)$	$G_a((v_i)_{i \in I}) = \text{true}$	$\text{read}(v_i)_{i \in I}$	guard
$I \subseteq \overline{1, n}$	$(\forall i \notin I : q_i = q'_i)$	$(v'_i)_{i \in I} = F_a((v_i)_{i \in I})$	$\text{write}(v'_i)_{i \in I}$	data transfer
<hr/>				
$(q_1, \dots, q_n) \xrightarrow{a}_\gamma (q'_1, \dots, q'_n)$				

We define $B = \gamma\pi(B_1, \dots, B_n)$ to be the composition of the atomic components $\{B_i\}_{i=1}^n$ where π is a partial order defined by the rule:

$$\frac{a \in \gamma, \quad q \in Q, \quad q' \in Q, \quad C : \text{boolean condition}}{q \xrightarrow{a}_\gamma q' \quad (\forall a' \in \gamma : \langle C \rightarrow a \prec a' \rangle \in \pi) \quad (\neg C \vee q \not\xrightarrow{a'}_\gamma)} \quad \frac{}{q \xrightarrow{a}_\pi q'}$$

We remind, that only one port from each component can participate to the same interaction. Moreover, different components have disjoint sets of variables. Each component i , non-deterministically, selects the transition that will lead to the successor state q'_i and consequently, the associated local variables (v_i, v'_i) to be modified. However, a global move is allowed only if the selected, for exchange, values (v_i, v'_i) (attached to the interacting ports p_i) satisfy the synchronization conditions (guard G_a and data-transfer function F_a).

The first rule corresponds to the firing of a transition $a \in \gamma$ for the obtained behavior $\gamma(B_1, \dots, B_n)$. A transition a is executed if all ports p_i are enabled and the guard G_a is true. The guard is evaluated on the current valuations. Once the transition is executed, a new valuation v'_i is computed as defined by F_a .

The second rule corresponds to the firing of a transition $a \in \gamma$ for the obtained behavior γ restricted by priorities. A transition a is executed if any other transition a' with higher priority than a is disabled or the state predicate C is false.

Example 5 Figure 2.6 illustrates the Precision Time Protocol (PTP) as the composition of the four components Master_clock, Slave_clock, Master_to_Slave and Slave_to_Master.

The g_{tick} interaction synchronizes all components by strongly connecting the tick ports. All other interactions are weak synchronizations. The execution of the interactions g_{inFol} , g_{outFol} , g_{inReq} and g_{outReq} is involved with data transfer between different components. g_1 , g_2 and g_3 are singleton connectors, i.e., each of them involves only a port. The priority π_1 disables the execution of g_{tick} if other interactions are available. Similarly, priorities π_2, π_3, π_4 and π_5 enforce the execution of g_{inFol} , g_{outSyn} , g_{outRep} and g_{inReq} respectively in case of conflict.

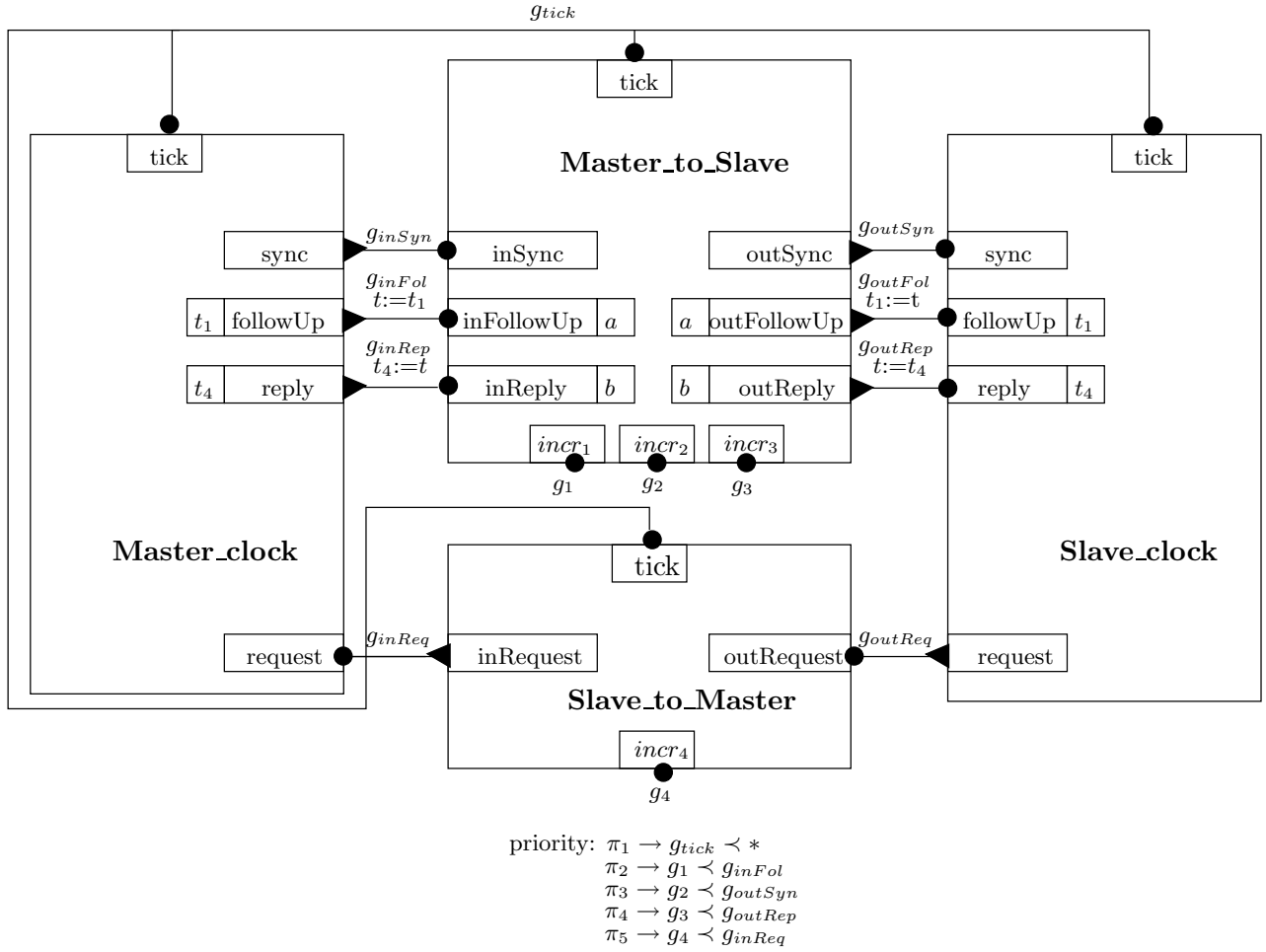


Figure 2.6: The PTP model as a composition of atomic BIP components

2.3 The BIP Language

The BIP language represents components of the BIP framework [12]. BIP language is a user-friendly textual language which provides syntactic constructs for describing systems. It leverages on C style variables and data type declarations, expressions and statements and provides additional structural syntactic constructs for defining component behavior, specifying the coordination through connectors and describing the priorities.

The basic constructs of the BIP language are the following:

- *atom*: to specify behavior, with an interface consisting of ports. Behavior is described as a set of transitions.
- *connector*: to specify the coordination between the ports of components, and the associated guarded actions.
- *priority*: to restrict the possible interactions, based on conditions depending on the state of the integrated components.
- *compound*: to specify systems hierarchically, from other atoms or compounds, with connectors and priorities.
- *model*: to specify the entire system, encapsulating the definition of the components, and specify the top level instance of the system.

Example 6 *The BIP description of the Master_clock atomic component of Figure 2.3 (left) is illustrated below:*

model PTP

port type DataPort (int i)

port type EventPort

atomic type Master_clock

```

export port EventPort tick=tick
export port EventPort sync=sync
export port EventPort request=request
export port DataPort followUp( $t_1$ )=followUp
export port DataPort reply( $t_4$ )=reply

```

```

place  $q_1, q_2, q_3, q_4$ 
initial to  $q_1$  do {}

```

```

on tick from  $q_1$  to  $q_1$ 
  do { $x++$ ;  $\theta_m++$ ;}
on sync from  $q_1$  to  $q_2$  (provided  $x==P$ )
  do { $x=0$ ;  $t_1=\theta_m$ }
on followUp from  $q_2$  to  $q_3$ 
on tick from  $q_3$  to  $q_3$ 
  do { $x++$ ;  $\theta_m++$ ;}
on request from  $q_3$  to  $q_4$ 

```

```

    do { $t_4=\theta_m$ }
  on reply from  $q_4$  to  $q_1$ 
end

```

Two types of ports are defined, DataPort and EventPort. A port type DataPort associates a port to an integer variable i . Variables associated to ports may be modified when executing the interaction in which the port participates. Ports followUp and request are instances of the type DataPort. A port type EventPort is an event port and it is not associated with any variable. The ports tick, sync and reply are instances of the type EventPort. All ports are exported at the interface of the component. Initially, the state of the component is at the place q_1 , the only place with token. The BIP code uses the constructs “on...from ...to” to represent transitions from one place to the other. The construct “provided” is used when the execution of a transition is restricted by a guard. Moreover, if the transition is associated with a function, the C code inside the constructs “do { ...}” is executed.

Components are composed by using connectors. A connector defines the set of possible interactions between ports of components and the corresponding data transfer between the variables associated with the ports. The BIP language allows the definition of connector types.

Example 7 Below is presented the syntax of four different types of connectors, RendezVousData, BroadcastData, RendezVousEvents and SingletonEvent connector.

```

connector type RendezVousData(DataPort in, DataPort out)
  define in out
  on in out
    down {out.x=in.x;}
end

```

```

connector type BroadcastData(DataPort in, DataPort out)
  define in' out
  on in
  on in out
    down {out.x=in.x;}
end

```

```

connector type RendezVousEvents(EventPort e1, EventPort e2)
  define e1 e2
  on e1 e2
  export port e
end

```

```

connector type SingletonEvent(EventPort e1)
  define e1
  on e1
  export port EventPort e
end

```

The RendezVousData connector defines a strong synchronization between two ports of type DataPort, in and out. The value x is copied from the port in to the port out each time the connector is executed. The BroadcastData connector defines a weak synchronization between the ports in and out of DataPort type. Port in initiates the synchronization. The RendezVousEvents

connector defines a strong synchronization between two ports of type `EventPort`. This interaction is exported to the environment through the `EventPort` *e*. The `SingletonEvent` connector involves only one port of type `EventPort`.

A *compound component* is a new component type defined from existing components by creating their instances, instantiating connectors between them and specifying the priorities. A compound offers the same interface as an atom, hence externally there is no difference between a compound and an atomic component.

Example 8 *The BIP description for the PTP compound component of Figure 2.6 is shown below:*

compound type `CompoundPTP`

```

component Master_clock masterCl
component Slave_clock slaveCl
component Slave_to_Master stmChannel
component Master_to_Slave mtsChannel

connector RendezVous4Events gtick(masterCl.tick, slaveCl.tick,
    mtsChannel.tick, stmChannel.tick)

connector BroadcastEvents ginSyn(masterCl.syn, mtsChannel.inSyn,)
connector BroadcastEvents goutSyn(mtsChannel.outSync, slaveCl.sync)
connector BroadcastEvents ginReq(masterCl.request, stmChannel.inRequest)
connector BroadcastEvents goutReq(stmChannel.outRequest, slaveCl.request)

connector BroadcastData ginRep(masterCl.reply, mtsChannel.inReply)
connector BroadcastData goutRep(mtsChannel.outReply, slaveCl.reply)

connector BroadcastData ginFol(masterCl.sync, mtsChannel.inSync)
connector BroadcastData goutFol(mtsChannel.outFollowUp, slaveCl.followUp)

connector SingletonEvent g1(mtsChannel.inc1)
connector SingletonEvent g2(mtsChannel.incr2)
connector SingletonEvent g3(mtsChannel.incr3)
connector SingletonEvent g4(stmChannel.incr4)

priority p1 gtick < *
priority p2 g1 < ginFol
priority p3 g2 < goutSyn
priority p4 g3 < goutRep
priority p5 g4 < ginReq

```

end

The four atomic components that constitute the PTP model are instantiated. For example component `Master_clock masterCl`, creates an instance of `Master_clock` component named

masterCl. Connectors are also instantiated, associating the ports of instantiated components through the interactions defined by the connector type. Finally, priorities are defined specifying an order between a pair of interactions.

2.4 The BIP Toolset

The BIP toolset provides tools for modeling, simulation, code generation and verifying BIP models. An overview of the BIP toolset is shown in Figure 2.7. The different components of the BIP toolset are presented below:

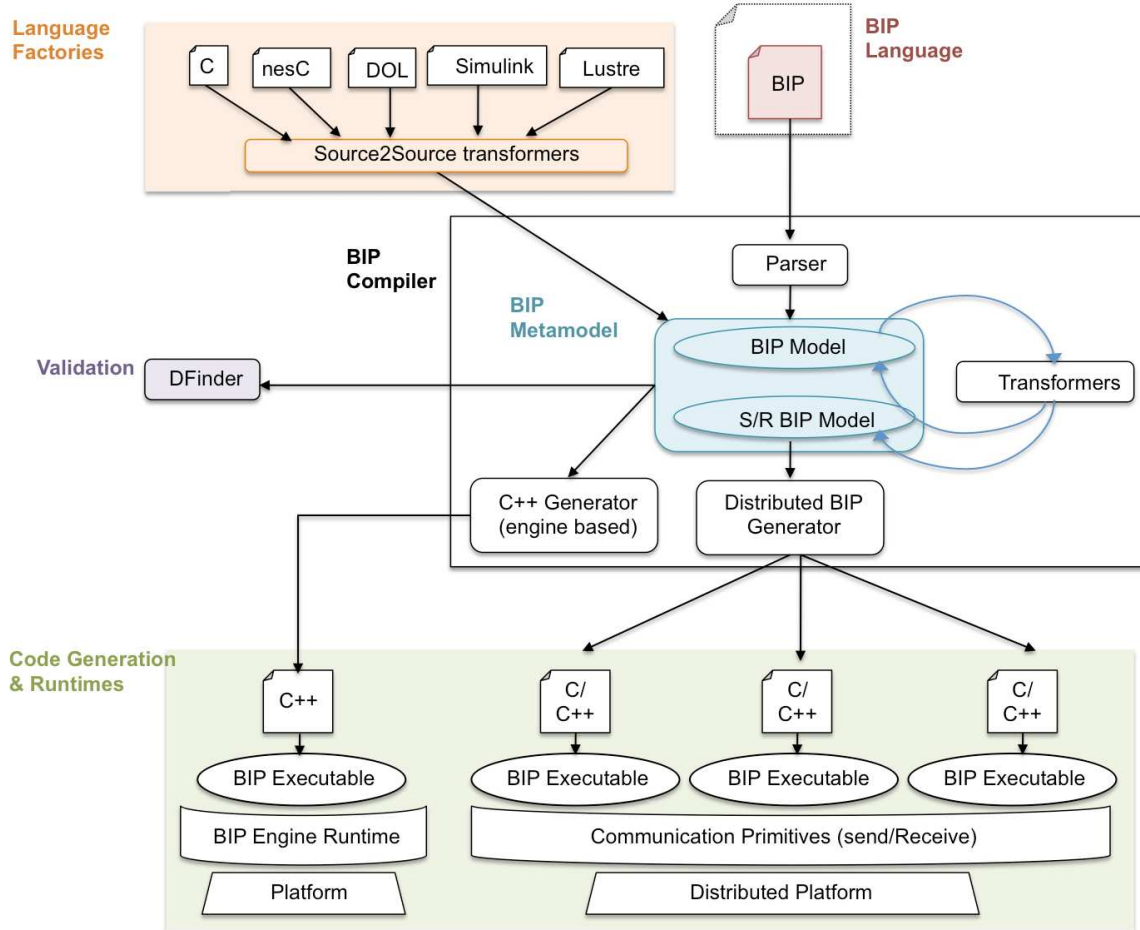


Figure 2.7: The BIP toolset

- *BIP Language:* It is used to define “types” (for components and connectors) and describe component architectures (assembly of instances of types).
- *Language Factories:* The application software includes various programming models. The translation of the application software into a BIP model allows its representation in a rigorous semantic framework. There exist several translations of several programming models into BIP, including LUSTRE [28], MATLAB/Simulink [66], AADL [32], GeNoM applications [13], NesC/TinuOS applications [14], C software and DOL systems [60].

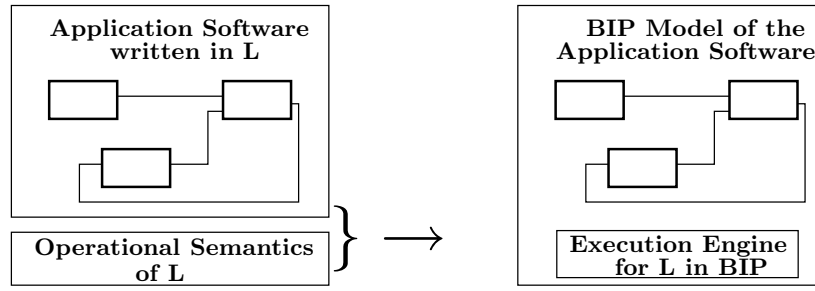


Figure 2.8: Translation method for a language L in BIP

There exist a general method for generating BIP models from a language L which involves three steps as shown in Figure 2.8. First, the translation of atomic components of the source language into BIP components. Second, the translation of coordination mechanisms between components of the application software into connectors and priorities in the target BIP model. Third, the generation of a BIP component modeling the operational semantics of the language.

- *BIP Compiler*: It is targeting the BIP Execution Engines. Both the generated code and the Engines are in C++.
- *BIP Metamodel*: It is used as the intermediate representation of BIP models. It has been used to implement model transformations.
- *Transformers*: The transformation of a BIP abstract system model into a concrete BIP system model (i.e. implementations) is obtained by expressing high level coordination mechanisms e.g., interactions and priorities by using primitives of the execution platform. This transformation usually involves the replacement of atomic multiparty interactions by protocols using asynchronous message passing (send/receive primitives) [48] and arbiters ensuring overall coherency e.g. non interference of protocols implementing different interactions. The transformations use a set of correct-by construction models and preserve functional properties. Moreover, they take into account extra functional constraints. There exist three types of transformations, architecture optimizations [26], distributed implementations [24] and memory management [27].
- *D-Finder*: It is a compositional verification tool for deadlock detection and generation of invariants. Verification is applied only to high level models for checking safety properties such as invariants and deadlock-freedom. To avoid inherent complexity limitations, the verification method applies compositionality techniques efficiently implemented by using heuristics in the D-Finder tool [15]
- *Code generation*: Monolithic C code is generated from sets of interaction components executed by the same processing unit. This transformation allows efficient implementation by avoiding overhead due to coordination between components.
- *Execution Engines*: They are middleware responsible for the coordination of atomic components, that is, they apply the semantics of the interaction and priority layers of BIP. Execution engines can be used for execution, simulation, statistical model checking, debug or state-space exploration (i.e. all traces) of BIP models. There are currently three engines available, single-thread [9], multi-thread [9] and real-time [6].

2.4.1 Code Generation for BIP Models

The code produced for BIP models is modular, that is, the code of atomic components is isolated from the glue code and the coordination code [25]. Glue code is the code produced for the data transfer on connectors and for priority evaluation between enabled interactions. Coordination code is the code orchestrating the whole execution. To achieve modularity, there is created a relatively simple interface for atomic components consisting of two functions *initialize* and *execute*:

- the *initialize* function is called once in order to initialize the component and to execute its behavior until the first stable state is reached. The function returns the set of ports on which the component is ready to interact together with their associated (up) values. This function correspond to the execution of the initial transition (**initial to ... do { ...}**);
- the *execute* function is called iteratively, after initialize. Its argument is one of the ports amongst the one previously proposed together with each associated value. This function performs the quantum of computation triggered by that port, starting from the current stable state and until the next stable state is reached. It returns the set of ports ready to interact. This functions corresponds to the execution of the transitions of the model, except from the initial.

There exist two main compilation flows for generating code from BIP tools, the direct compilation of *Send/Receive BIP* models and the *engine-based* compilation.

The *Send/Receive BIP* compilation can be used to generate distributed implementations from BIP models. The transformation of BIP models into Send/Receive models consists of three steps. First, breaking atomicity of actions in atomic components by replacing strong synchronizations with asynchronous Send/Receive interactions. Second, inserting several distributed Engines that coordinate execution of interactions according to a user-defined partition. Third, augmenting the model with a distributed algorithm for handling conflicts between distributed Engines.

In the *engine-based* compilation, the generated code needs an engine for its execution. It can be used for targeting non-distributed platforms. There has been developed a C++ code generator for BIP programs that supports the full BIP syntax. The following BIP Engines are currently available, *Single-Thread Engine*, *Multi-Thread Engine* and *Real-time Engine*.

Single-Thread Engine Implementation

From a BIP model, a compiler is used to generate C++ code for atomic components and glue. The code is then orchestrated by a sequential engine that interprets the BIP operational semantics rules. The architecture of the sequential implementation is shown in Figure 2.9 and the main algorithm is presented in Figure 2.4.1.

The algorithm starts by initializing and retrieving the set of enabled ports for each atomic component. In the main loop, the engine computes from the set of ports offered by individual components and defined by connectors, the set of the enabled interactions. Amongst these, it chooses a maximal one, according to priorities. For the chosen interaction, the engine executes the data transfer followed by the specific computations of all involved atomic components.

The centralized engine has run-time options for execution and enumerative state-space exploration. In *execution* mode, the engine offers possibilities of running either a random trace or an interactive trace. In the *state-space* exploration mode, the engine generates the underlying labeled transition systems (LTS) of the model, corresponding to the semantics of the model.

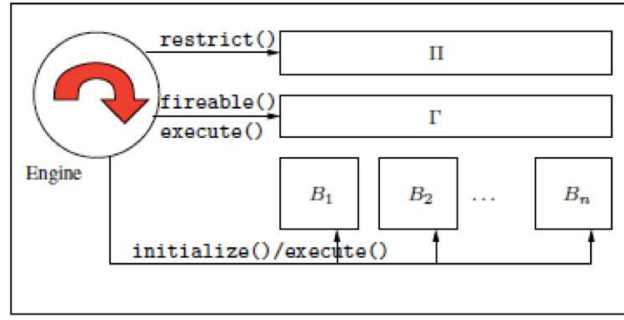


Figure 2.9: Architecture of sequential implementation

```

foreach j in 1,n do
   $P_j := B_j.\text{initialize}()$ ;
do forever
   $A := \text{compute-fireable}(\Gamma, P_1, \dots, P_n)$ ;
   $A^{max} := \text{restrict-priorities}(\Pi, A)$ ;
  if  $A^{max}$  is not empty then
    choose  $a = (p_i)_{i \in I}$  in  $A^{max}$ ;
     $\text{execute-data-transfer}(a)$ ;
    foreach  $i$  in  $I$  do
       $P_i := B_i.\text{execute}(p_i)$ ;
  else
     $\text{deadlock}()$ ;
    stop;
  fi;
done

```

Figure 2.10: Algorithm of the engine for sequential implementation

Multi-Thread Engine Implementation

The principle of multi-threaded implementation with centralized engine is illustrated in Figure 2.11 and the algorithm for respectively atomic components and engine is presented in Figure 2.12. This implementation is based on the notion of partial state semantics [10] where interactions are allowed to fire as soon as only the involved components are stable.

Each atomic component is assigned to a different thread (processor), the engine being assigned a thread as well. Each atomic component performs its computations locally and then, when it reaches a stable state, it notifies the engine about the ports on which it is willing to interact. The engine is parametrized by an oracle. Iteratively, the engine computes feasible interactions available on stable components. Then, if such interactions exist and the oracle allows them, the engine selects one for execution and notifies the involved components.

2.5 Discussion

The BIP (Behavior, Interaction, Priority) component framework is a formalism for modeling heterogeneous component-based systems. It allows the description of systems as the composition

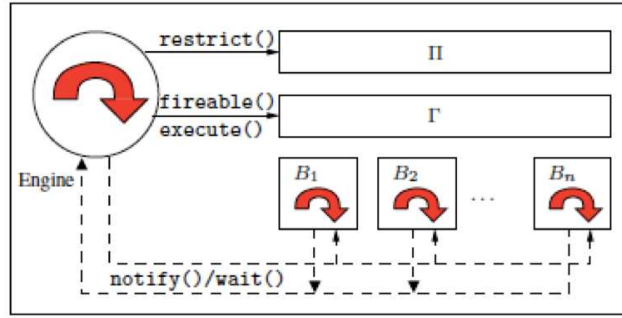


Figure 2.11: Architecture of distributed implementation

of generic atomic components characterized by their behavior and their interfaces. It supports a system construction methodology based on the use of two families of composition operators: interactions and priorities. Interactions are used to specify multiparty synchronizations between components as the combination of two protocols: rendezvous (strong symmetric synchronizations) and broadcast (weak asymmetric synchronizations). Priorities between interactions are used to restrict non determinism inherent to parallel systems. They are particularly suited for modeling scheduling policies.

BIP characterizes systems as points in three-dimensional space: *Behavior* \times *Interaction* \times *Priorities* as represented in Figure 2.13. Elements of the *Interaction* \times *Priority* space characterize the overall architecture. Each dimension, can be equipped with an adequate partial order, e.g. refinement for behavior, inclusion of interactions, inclusion of priorities. Some interesting concepts of this representation are the following:

- Any combination of behavior, interaction and priority models meaningfully defines a component. Separation of concerns is essential for defining a component's construction process as the superposition of elementary transformations along each dimension.
- Different subclasses of components e.g., untimed/timed, asynchronous/synchronous, event-triggered/data-triggered, can be unified through transformations in the construction space. These transformations often involve displacement along the three coordinates.
- The component construction space provides a basis for the study of architecture transformations allowing preservation of properties of the underlying behavior. The characterization of such transformations can provide (sufficient) conditions for correctness by construction such as compositionality and composability results from deadlock-freedom.

In contrast to existing formal frameworks, BIP is expressive enough to directly model any coordination mechanism between components. It has been successfully used to model complex systems including mixed hardware/software systems and complex software applications like the DALA robot [1], the Heterogeneous Communication System (HCS) [11], the NesC/TinyOS applications [14] and DOL systems [60]. $\ddot{\gg}i$

```

Pi := initialize();
do forever
  notify(E, Pi);
  wait(E, pi);
  Pi := execute(pi);
done

foreach j in 1..n do
  Pj := ⊥;
do forever
  A := compute-fireable( $\Gamma$ , P1, ...Pn);
  Amax := restrict-priorities( $\Pi$ , A, O);
  if Amax is not empty then
    choose a = (pi)i ∈ I in Amax;
    execute-data-transfer(a);
    foreach i in I do
      notify(Bi, pi);
      Pi := ⊥;
    else
      break;
  fi
done
if forall j = 1..n. Pj ≠ ⊥ then
  deadlock();
stop;
fi
done

```

Figure 2.12: The algorithms for atomic components (left) and engine (right)

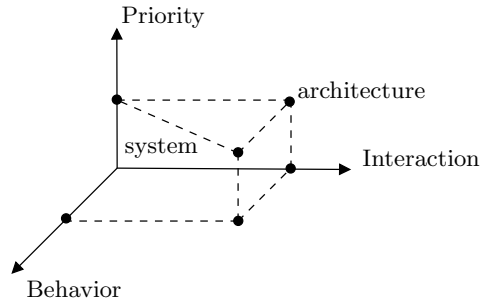


Figure 2.13: The Construction Space

Chapter 3

Synchronous Formalisms

The history of synchronous languages dates back to the early 1980's [42]. Three French projects started independently aiming at designing the three programming languages ESTEREL [21], SIGNAL [20] and LUSTRE [43]. Other languages like SML [46] and STATECHARTS [45] were developed in the same time adopting some aspects of the synchronous model. However, these languages were not designed to be used for programming. SML is a hardware description language and STATECHARTS is a specification language.

Nowadays, there exist numerous languages and formalisms that rely on the synchronous principles (see Chapter 1). They can be classified in three categories, *imperative*, *declarative* and *graphical formalisms*. *Imperative* programming describes computation in terms of statements that as they change they modify the state of the program. An imperative program introduces memory states that are modified each time the actions of the program change. An imperative program is a sequence of such actions also called instructions. Examples of imperative synchronous languages are the ESTEREL [21] language, the Synchronous Data Flow (SDF) language [52], the Synchronous Structures formalism [59] and some of the MoC in Ptolemy [38].

Declarative programming expresses the logic of a computation without describing its control flow. Examples of declarative languages are the languages LUSTRE [43], SIGNAL [20], N-Synchronous [33] and the 42 [55].

Graphical formalisms are based on automata, petri nets, blocks diagrams or other representation to describe the specification and design of systems. Some examples are SyncCharts [8], MarkedGraphs [36], ARGOS [56], StateCharts [45] and MATLAB/Simulink [2].

This chapter is structured as follows. Section 3.1 describes the LUSTRE language. It presents the different types of operators (single-clock and multi-clock) and gives some references for static verification and code generation. Section 3.2 describes the SIGNAL synchronous language. Section 3.3 describes the MATLAB/Simulink framework. The description is restricted to the discrete-time subset of Simulink. Conclusions are drawn in section 3.4.

3.1 The LUSTRE Language

LUSTRE is a dataflow synchronous language for programming reactive systems. LUSTRE programs operate on flows of values, that are infinite sequences $(x_0, x_1, \dots, x_n, \dots)$ of values at logical time instants $(0, 1, \dots, n, \dots)$. An abstract syntax for LUSTRE programs is shown below. *In* (resp. *Out*) denotes the set of inputs (resp. output) of a node. Symbols N , E , x , v , b denote respectively node names, expressions, flows, constant values and Boolean values.

```

program ::= node+
node    ::= node N (In) (Out) equation+
equation ::= x = E |
            x, ..., x = N(E, ..., E)
E       ::= x | v | op(E, ..., E) | pre(E, v) |
            E when b | current E

```

A LUSTRE program is structured as a set of *nodes*. Each node computes output flows from input flows. Output flows are defined either directly by means of equations of the form $x = E$, meaning $x_n = E_n$ for any time instant $n \geq 0$ or, as the output of other already defined nodes of the form $x, \dots, x = N(E, \dots, E)$. Each flow (and expression) is associated with a logical clock. Implicitly, there always exist a unique, fastest, *basic clock* which defines the step (or basic clock cycle) of a synchronous program. Depending on this clock, other slower clocks can be defined as the sequences of time instants where Boolean flow values take the value true.

LUSTRE has only few elementary basic types: boolean, integer and one type constructor: *tuple*. Complex types can be imported from a host language. Constants in LUSTRE are those of the basic types and those imported from the host language and their clock is the basic one. Usual operators over basic types are available such as arithmetic, boolean, relational and conditional. Functions can be imported from the host language. These are *combinatorial operators* (*op*) and the unit delay **pre** operator known as *single-clock operators*. They operate on operands that share the same clock. Besides these operators, LUSTRE has operators which operate on multiple clocks. These are the **when** and the **current** operator known as *multi-clock operators*.

3.1.1 Single-clock Operators

Single-clock operators contain *constants*, *basic combinatorial operators* and the *unit delay operator*. Flows of values that correspond to *constants* are constant sequences of values. *Combinatorial* (memory-less) operators include usual Boolean, arithmetic and relational operators. The unit delay **pre** operator gives access to the value of its argument at the previous time instant. For example, the expression $E' = \mathbf{pre}(E, v)$ means $E'_0 = v$ and $E'_i = E_{i-1}$, for all $i \geq 1$.

Example 9 Figure 3.1 shows a discrete integrator written in LUSTRE (left). On Figure 5.4 is illustrated the synchronous network of operators for this example. It uses the single-clock operators “+” and **pre**.

```

node Integrator(i: int)
  returns o: int;
let o = i + pre(o, 0); tel;

```

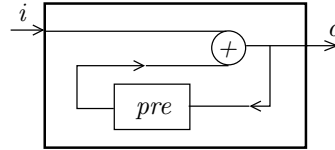


Figure 3.1: An integrator described in LUSTRE

The Integrator has input and output flows, i and o respectively, both of type integer and which operate on the basic clock. The output flow o is obtained by adding to its previous value $\mathbf{pre}(o, 0)$ the input flow i . The equation of the integrator is the arithmetic operation “+” between a flow and the expression **pre**. The expression $\mathbf{pre}(o, 0)$ gives access to the value of o at the previous time instant and is initialized to zero. The instants of a possible execution are shown in Figure 3.2.

<i>basic clock</i>	1	2	3	4	5	6	7	...
i	2	5	-7	0	3	9	1	...
pre	0	2	7	0	0	3	12	...
o	2	7	0	0	3	12	13	...

Figure 3.2: Execution instants for the Integrator node of Example 9

Example 10 The example in Figure 3.3 is a version of a watchdog device that monitors response times [44]. It receives three events, set, reset and deadline. It outputs the alarm event. Events are represented by boolean variables and are present when their values are true. An alarm event occurs whenever both the deadline event and the set event are present. The `is_set` is a local boolean variable, initially evaluated to set. The `is_set` becomes true if set is true, it becomes false if reset is true, otherwise it keeps the same value as in its previous evaluation. Figure 3.4 shows execution instants for this node.

```

node Watchdog(set, reset, deadline: bool)
  returns alarm: bool;
var is_set: bool;
let
  alarm = deadline and is_set;
  is_set = set -> if set then true
                  else if reset then false
                  else pre(is_set);
tel.

```

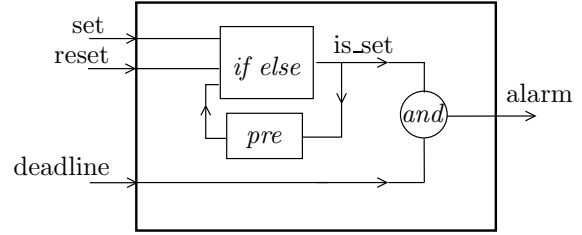


Figure 3.3: A watchdog described in LUSTRE

<i>basic clock</i>	1	2	3	4	5	...
set	true	false	false	true	false	...
reset	true	true	false	false	false	...
deadline	false	true	false	true	true	...
is_set	true	false	false	true	true	...
pre (is_set)	nil	true	false	false	true	...
alarm	false	false	false	true	true	...

Figure 3.4: Execution instants for the watchdog LUSTRE node of example 10

3.1.2 Multi-clock Operators

In order to define and manipulate flows operating on slower clocks, LUSTRE provides two additional operators. The *sampling* operator **when**, samples a flow depending on a Boolean flow. The expression $E' = E$ **when** b , is the sequence of values E when the Boolean flow b is true. The expression E and the Boolean flow b have the same clock, while the expression E' operates on a slower clock defined by the instants at which b is true. The *interpolation* operator **current**, interpolates an expression on the clock which is faster than its own clock. The expression $E' = \mathbf{current} E$, takes the value of E at the last instant when b was true, where b is the Boolean flow defining the slower clock of E .

Example 11 Example of using sampling and interpolating operators is shown in Figure 3.5.

The basic clock defines six clock cycles. The Boolean flow b and the flow x operate on the basic clock. The flow b defines a slower clock operating at the cycles 3, 5 and 6 of the basic clock. These are the instants that the value of b is true.

The sampling operator **when** defines the flow y that operates on the slower clock b . The flow y is evaluated when b is defined, that is, at the clock cycles 3, 5 and 6.

The interpolation operator **current** produces the flow z on the basic clock. The flow z has the same clock with b . For the first two instances the value of z is undefined because y is evaluated for first time at the clock cycle 3. For any other instant, if b is true, the value of z is evaluated to y . Otherwise, it takes the value of y at the last instant when b was true. For instance, at the clock cycle 3, the slower clock b is defined and the value of z is evaluated to the current value of y , that is x_3 . For clock cycle 4, the slower clock b is not defined and the value of z takes x_3 , that is the value of z at the last instant b was true.

is the value of z_3 , that is, the last time when b was true.

basic clock	1	2	3	4	5	6	...
b	false	false	true	false	true	true	...
x	x_1	x_2	x_3	x_4	x_5	x_6	...
$y = x$ when b			x_3		x_5	x_6	...
$z =$ current y	nil	nil	x_3	x_3	x_5	x_6	...

Figure 3.5: Example of use of *when* and *current* multi-clock operators

Example 12 The following LUSTRE node example describes an `input_output_handler`.

```
node input_handler(a: bool, x: int when a)
returns y: int;
let y = if a then current x else pre(y, 0);
tel ;
```

```
node output_handler(c: bool, y: int)
returns z: int when c;
var yc: int when c;
let yc = y when c; z = yc * yc ;
tel ;
```

```
node input_output_handler(a,c: bool,
                          x: int when a)
returns z: int when c;
var y: int;
let y = input_handler(a, x);
    z = output_handler(c, y);
tel;
```

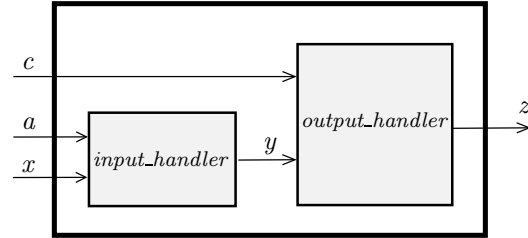


Figure 3.6: An `input_output_handler` described in LUSTRE

Figure 3.6(right) illustrates the network corresponding to the `input_output_handler` LUSTRE node (figure 3.6 (left)). It reads three inputs, c , a and x and on the internal nodes `input_handler` and `output_handler` it computes the output z .

It shows the contents of the `input_output` node which are the subnodes `input_handler` and `output_handler`, the inputs `c`, `a` and `x` and the output `z`.

This LUSTRE program consists of three nodes, the `input_handler`, the `output_handler` and the `input_output_handler` which is the main node of the program. The `input_handler` receives a Boolean flow `a` on the basic clock and an integer flow `x` when `a` is true. It produces an integer flow `y` at every cycle of the basic clock by interpolating the value of `x`. The `output_handler` receives the Boolean flow `c` and the integer flow `y`, both at every cycle of the basic clock. It samples `y` and produces the output flow `z` when `c` is true. The `input_output_handler` node interconnects the two previous nodes. It receives two Boolean flows `a` and `c` at every clock cycle and an integer flow `x` when `a` is true. The internal flow `y` produces at each clock cycle the most recent value of `x`. Finally, the `input_output_handler` node produces the output flow `z` when `c` is true using the most recent available value of `y`. Note that the variables `a` and `c` represents slower clocks, independent from each other. Figure 3.7 shows execution instants for the `input_output_handler` LUSTRE node of Figure 3.6.

basic clock	1	2	3	4	5	6	7	8	...
<i>a</i>	false	true	true	true	false	false	true	false	...
<i>c</i>	true	false	true	true	false	false	true	true	...
<i>x</i>		x_1	x_2	x_3			x_4		...
<i>y</i>	0	x_1	x_2	x_3	x_3	x_3	x_4	x_4	...
<i>z</i>	0		$x_2 \times x_2$	$x_3 \times x_3$			$x_4 \times x_4$	$x_4 \times x_4$...

Figure 3.7: Execution instants for the `input_output_handler` LUSTRE node of Figure 3.6

Example 13 Figure 3.8 shows a multiplier (`mux`) LUSTRE node. It reads a variable `m` at each clock cycle and produces three outputs; `y` and `c` are produced at each cycle of the basic clock and `x` when `c` is true. If the boolean variable `c` is false, `y` decreases its value by one. When `y` is evaluated to zero, the value of `c` becomes true and `x` produces the current value `m`. Figure 3.9 shows executions instants for the `mux` example of Figure 3.8.

```

node mux(m: int)
  returns (c: bool;
          x: int when c; y: int);
let
  y = if c then current x else pre y-1;
  c = true -> pre y=0;
  x = m when c;
tel.

```

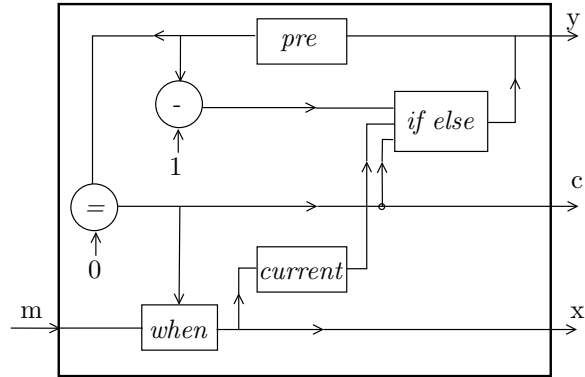


Figure 3.8: A `mux` LUSTRE node

3.1.3 LUSTRE Compiler and Code Generation

The LUSTRE compiler guarantees that the system under design is deterministic and conforms to the properties defined by the synchronous hypothesis [69]. It accomplishes this task thanks to static verification which is summarized in the following steps:

<i>basic clock</i>	1	2	3	4	5	6	7	8	...
m	5	9	4	3	12	2	3	5	...
c	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	...
m when c	5	-	-	-	-	-	3	-	...
y	5	4	3	2	1	0	3	2	...

Figure 3.9: Instants of the execution for the mux example

- *Definition checking*: every local and output variable should have one and only one;
- *Clock consistency*: every operator is applied to operands on suitable clocks;
- *Absence of cycles*: any cycle should use at least one **pre** operator.

LUSTRE compiler provides two methods for generating code. The first method is the code generation for a *mono-processor and mono-thread implementation*. The compiler generates monolithic endless single loop C code. The body of this code implements the inputs to outputs transformations at one clock cycle. The generation of C code is done in two steps:

1. introduce variables for implementing the memory needed by the **pre** operators and
2. sort the equations in order to respect data-dependencies.

The second method concerns the *distributed implementation*. LUSTRE programs can be deployed on a distributed architecture via the object code (OC) automaton-format [31]. This technique is closely related to the Kahn process networks. The distribution of an OC code is based on the assumption that there exists a set $\{s_1, \dots, s_n\}$ of execution sites (processors) and that each of these sites is associated with an action of the OC automaton. For a LUSTRE program, each variable of the main node is assigned to an execution site. Propagating this assignment inside internal nodes provides a site assignment for each variable in the expanded program. The basic idea of the method of the distribution for a LUSTRE program contains four steps:

1. The code of the automaton is replicated on each site;
2. On each replication, the instructions that do not concern the considered site are erased;
3. For any pair (s_i, s_j) of sites, the order that s_i computes its own variables and the order in which s_j uses these variables is known. Thus, statements for communicating values computed by s_i and used by s_j can be introduced without introducing deadlocks;
4. Auxiliary “dummy” communications are added for synchronization.

3.2 SIGNAL

SIGNAL is a declarative synchronous language for real time programming [18, 51]. It relies on constructs which can be combined using a composition operator. These constructs describe processes and involve signals. A *signal* is an infinite typed sequence of data. The status of a signal can be either *present* or *absent* (denoted by \perp). The *data* of a signal can be of a standard type like boolean, real and integer or of a specific type like *event type*. An *event* signal x , with syntax **event** x , is *true* if and only if x is present, otherwise is equal to \perp .

Each signal has an associated *clock*. Signals that are present simultaneously have the same clock. Signals and clocks are related through *equations*. Equations are built on operators. *Operators* can be of two types, single-clocked and multi-clocked. Single-clocked operators involve signals which have the same clock. Operators that involve signals with different clocks are multi-clocked operators. A *process* is defined by an equation or a composition of equations. Parallel composition of processes relates signals and their corresponding activation clocks.

The following sections present the basic relations on clocks and the SIGNAL operators.

3.2.1 Clock Relations

SIGNAL defines operations on clocks of signals. For signals x , y , some operations on clocks are presented below:

- *Clock of a signal*: $h := \wedge x \equiv \text{if } x = \perp \text{ then } \perp \text{ else } true$
- *Clock selection*: **when** b , it returns the clock that represents the implicit set of instants at which the signal b is true. It is denoted by $[b]$.
- *Synchronization*: $x \wedge y$, it means that the signals x and y have the same clock such that $(|x \wedge y|) = (|h := (\wedge x = \wedge y)|)$
- *Clock product*: $x \wedge * y$ specifies the clock intersection of signals x and y such that $(|x \wedge * y|) = (|\wedge x \text{ when } \wedge y|)$
- *Clock union*: $x \wedge + y$ specifies the clock union of x and y iff x or y is present such that $(|x \wedge + y|) = (|\wedge x \text{ default } \wedge y|)$
- *Clock order restriction*: $x \wedge < y$ specifies the restriction of x not to be more frequent than y such that $(|x \wedge < y|) = (|x \wedge = x \wedge * y|) = \wedge x \text{ when event } \wedge y$.

3.2.2 Single-clocked operators

Combinatorial

A combinatorial operator can be a classical arithmetic (+, -, /, *, ...) or logical (**and**, **not**, **<**, **>**, ...) operator. It produces an output signal z computed on two input signals x and y . A combinatorial operator, with syntax

$$z = x \text{ op } y$$

defines a process such that $z_t \neq \perp \Leftrightarrow x_t \neq \perp \Leftrightarrow y_t \neq \perp$. The behavior of an operator, for instance the + operator such that $z = x + y$ is illustrated on the following table:

x :	2	5	1	0	4	1	3	7	...
y :	0	4	3	6	6	7	1	0	...
z :	2	9	4	6	10	8	4	0	...

Delay

The delay operator defines a signal y whose t th element is the $(t-1)$ th element of its input x . At the first instant, it takes an initialization value c . The delay operator, with syntax

$$y := x\$1 \text{ init } c$$

defines a process such that $y_t \neq \perp \Leftrightarrow x_t \neq \perp$, $\forall t > 0 : y_t = x_{t-1}, y_0 = c$. The behavior of the delay operator with initial condition $c = 0$ is illustrated in the following table:

$x :$	2	5	1	0	4	1	3	7	...
$y :$	0	2	5	1	0	4	1	3	...

3.2.3 Multi-clocked operators

Under-sampling

The under-sampling operator delivers a signal y whenever the data input signal x and the boolean input signal b are present and b is true. The under-sampling, with syntax

$$y := x \text{ when } b$$

defines a process such that $y_t = x_t$ if $b_t = \text{true}$ else $y_t = \perp$. The behavior of the under-sampling operator is illustrated in the following table:

$x :$	1	2	\perp	3	4	\perp	5	6	...
$y :$	<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>	\perp	<i>false</i>	...
$z :$	1	\perp	\perp	\perp	4	\perp	\perp	\perp	...

Deterministic merging

The deterministic merging operator defines a signal z by merging two signals x and y with priority to x when both processes are present simultaneously. The deterministic merging, with syntax

$$z := x \text{ default } y$$

defines a process such that $z_t = x_t$, if $x_t \neq \perp$, else $z_t = y_t$, if $y_t \neq \perp$. The behavior of the default operator is illustrated in the following table:

$x :$	1	2	\perp	3	4	\perp	5	\perp	...
$y :$	\perp	\perp	3	4	10	8	9	2	...
$z :$	1	2	3	3	4	8	5	2	...

3.2.4 Parallel Composition

For P and Q two processes, the composition of P and Q , written $(P \mid Q)$, defines a new process where common names refer to common signals. Then the processes P and Q communicate through their common signals.

3.2.5 An example

The following example illustrates a SIGNAL program.

Example 14 Consider a program that reads an input IN , the value of which is decreased at each step by one until it becomes ≤ 0 . The syntax of this program is shown below:

```
(  X := IN default ZX - 1
  | ZX := X$1 init 0
  | B := (ZX ≤ 0)
  | IN ^ = when B
  | H ^ = B ^ = X ^ = ZX
)
```

The first equation says that X is equal to IN , if IN is present, otherwise it is equal to $ZX - 1$. According to the third and fourth equations, IN is present **when** B , that is, when $(ZX \leq 0)$. If this condition does not hold, then the value of X is equal to its previous value, decreased by one. The last equation shows equality on the clocks of H, B, X and Z .

3.3 MATLAB/Simulink

MATLAB/Simulink [2] is a very popular commercial tool for model-based design and simulation of dynamic embedded systems. Simulink systems are represented graphically using blocks and communication links for communication between blocks.

Example 15 Figure 3.10 shows a Simulink© model. This is a model for the anti-lock breaking system of a car. It simulates the dynamic behavior of a vehicle under hard braking conditions. Blocks such as Weight, gain, Add, Saturation, Scope and Terminator are atomic Simulink blocks and are dedicated to perform some specific operations. The blocks WheelSpeed and RelativeSleep are subsystems, that is, they are constructed incrementally from atomic blocks and other subsystems. Communication links are directed arcs connecting outputs to inputs of different blocks.

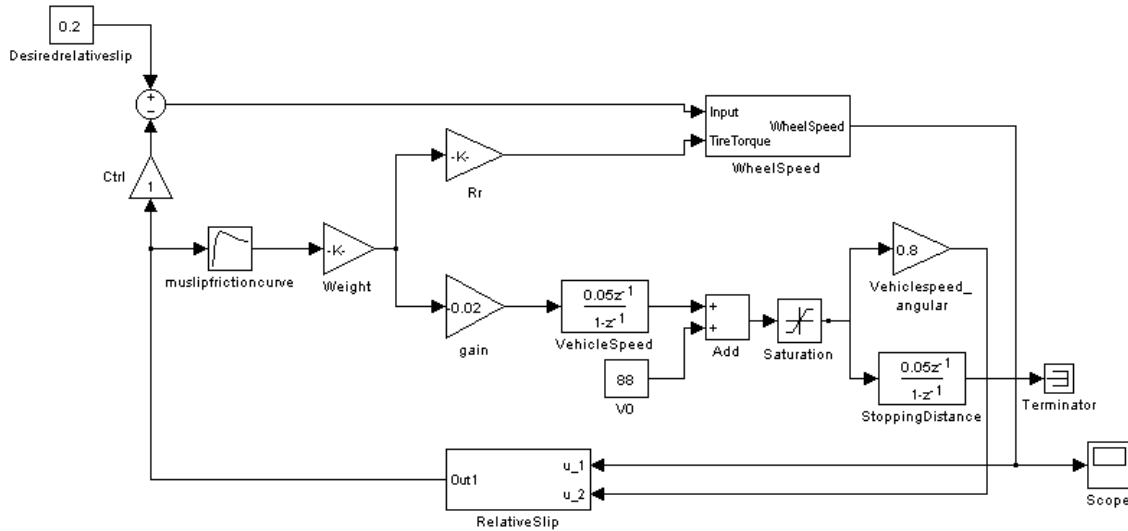


Figure 3.10: Anti-Lock Braking (ABS) Simulink© model

Simulink is widely used by engineers since it provides a wide variety of block libraries for implementing and testing discrete and continuous system. It is also used for research and educational purposes. Simulink offers a wide variety of *simulation parameters*, like simulation time, solver options, tolerance and step size. In this section we restrict the description of Simulink on discrete-time models of Simulink which can be simulated using “*fixed-step solver in single tasking mode*”.

3.3.1 Signals

Models described in the discrete-time fragment of Simulink operate on discrete-time signals, that are, piecewise-constant functions defined on the time domain $\mathbb{R}_{\geq 0}$ and with values on an arbitrary data domain (usually, a fixed power set \mathbb{R}^k).

Simulink models define transformations on discrete-time signals by means of structured block diagrams. These diagrams are constructed hierarchically from atomic blocks, defining elementary transformations (e.g., delay, sampling, arithmetic, etc.), and dataflow links, expressing instantaneous data communication.

Every signal s in a discrete-time Simulink model is characterized by its sample time, that is, the period $k > 0$ of time at which the signal can change its value. Hence, a signal s can change its value only at integer multiples¹ of k , and remains unchanged within every left-closed right-open interval $[n \cdot k, (n + 1) \cdot k[$, for $n \in \mathbb{N}$.

In Simulink models, the sample time of signals can be either explicitly provided by the modeler e.g., as an annotation to atomic blocks, or left unspecified. In the latter situation, the sample time is *inherited*, that means, inferred from the sample times of related signals using Simulink specific inference rules.

3.3.2 Ports and Atomic Blocks

Data ports

Simulink uses *inports* and *outports* to define dataflow connection endpoints in subsystems. They are used to transfer signals between the subsystems and their environment. The sample time of the ports defines the period in which the signal is updated (i.e. read or written). The inports and outports are graphically represented as shown in Figure 3.11.

Control ports

Simulink uses control ports to produce triggering events (*trigger port*) or to provide enabling conditions (*enable port*) for the execution of subsystems. Figure 3.11 shows the graphical notation for the two types of control ports.

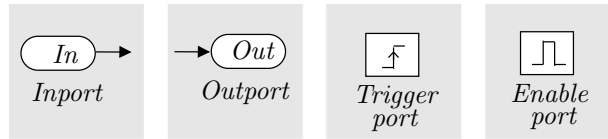


Figure 3.11: Data ports and control ports in Simulink

A *trigger port* produces an event that activates the execution of a triggered subsystem depending on some condition on an incoming signal. In Simulink, this condition can be either *rising*, *falling* or both. For example, in case of *rising* the activation event is produced when the input signal rises from a negative or zero value to a positive value.

An *enable port* defines a condition for the execution of an enabled subsystem depending on an incoming signal. In Simulink, the enabling condition holds as long as the value of the incoming signal is greater than zero. The enable port specifies one of the two states when it executes after being disabled, *held* or *reset*, depending if it holds the previous values of the subsystem or resets to the initial conditions.

Example 16 Figure 3.12 shows the timing diagram of a signal. A *rising* trigger signal occurs at time steps 2, 4 and 7. An enabling signal occurs between the time steps 4 and 6.

Sources and Sinks

Source blocks produce signals according to some patterns and with a specified or inherited sample time. Some examples are the Pulse Generator, the Sine Wave, the Constant blocks (see figure 3.13(a)).

¹Simulink allows as well for an offset, however for the sake of simplicity we always consider the offset equal to zero.

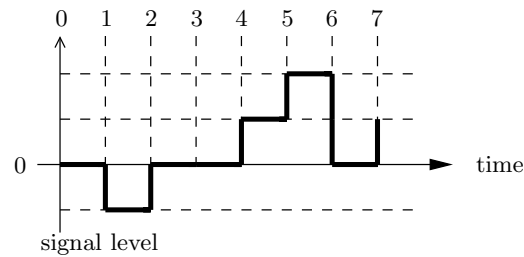


Figure 3.12: Timing diagram of a signal in Simulink

Conversely, sink blocks “consume” signals. Some examples are the *Scope* block which is used to display graphically one or more input signals and the **simout** for writing signal data to the MATLAB workspace (see Figure 3.13(b)).

Combinatorial blocks

Combinatorial blocks combine one or more input signals and produce one (or more) output signal(s) as the result of an instantaneous operation. The sample times of all input and output signals are equal. Some examples of combinatorial blocks provided by Simulink are usual arithmetic operators, relational operators, Boolean operators, switches, saturation blocks, lookup tables (see Figure 3.13(c)).

Unit delay

A *unit-delay* block delays the input signal for one period of the (input) sample time. During the first period, the unit-delay produces a user-specified constant signal value. This block may also perform a sample time change between the input and the output signals as follows: the sample time of the output can be smaller than (i.e., strict integer divisor of) the sample time of the input signal (see Figure 3.13(d)).

Zero-order hold

A *zero-order hold* block acts as a sampler. It holds the output constant for one period of the (output) sample time with the latest value of the input. This block may also perform a sample time change between input and output signals as follows: the sample time of the output can be greater than (i.e., strict integer multiple of) the sample time of the input signal (see Figure 3.13(d)).

Transfer functions

A *transfer function* block transforms an input signal according to a given discrete-time transfer function. The sample time of the input and output signals are equal (see Figure 3.13(e)).

3.3.3 Subsystems

Subsystems are user-defined assemblies constructed recursively from atomic blocks and other subsystems. They are used to encapsulate some reusable functionality, that can be plugged (i.e. called) in a system model or other subsystems.

The communication between a subsystem and its calling environment is realized through data ports. Data ports are inside the subsystem, exported at its interface. They are used to

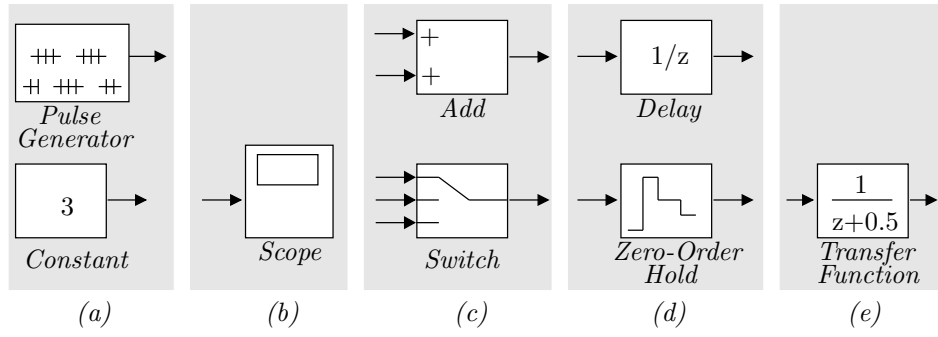


Figure 3.13: Atomic blocks in Simulink

convey signals, produced outside (resp.inside), towards (resp. outwards) the subsystem.

Simulink does not impose any syntactical restrictions on the inner blocks of the subsystems. However, type checking and sample type checking rules are applied to ensure consistency of computations e.g., the GCD rule for combinatorial operators [70]. The GCD (greatest common divisor) rule states that the output of a block will have as sample time the GCD of the sample times of the inputs. For instance if the two inputs of an *Add* block have 4 and 9 sample times respectively, then the sample time of the output will be 1.

In addition, there exists some support for controlled execution of subsystems. Simulink offers two basic mechanisms: *trigger* conditions, that can be used to activate triggered subsystems for execution and *enabling* conditions, that are used to enable/disable the execution of a subsystem.

Triggered Subsystems

Triggered subsystems execute instantaneously only when a trigger event occurs. Trigger events are defined as the rising or falling (or both) of a signal defined outside the subsystem.

Triggered subsystems do not have explicit sample time i.e., since their execution is triggered by data-change events, it is not directly time dependent. Simulink requires that all blocks within triggered subsystems have inherited sample time. Consequently, triggered subsystems contain only atomic blocks and triggered subsystems (not continuous time subsystems, not enabled subsystems).

Example 17 Figure 3.14 illustrates a Simulink model which contains a Triggered Subsystem.

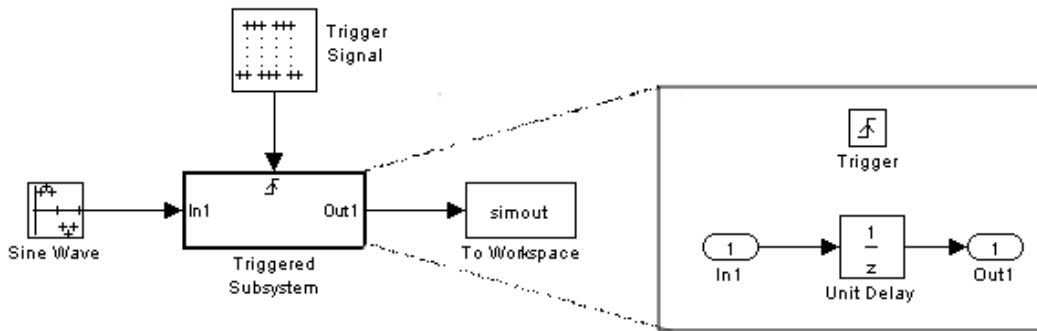


Figure 3.14: A Simulink model which contains a triggered subsystems

The Triggered Subsystem contains a Unit Delay block, an inport In1 and an output Out1, it reads inputs from the Sine Wave block and sends outputs to the simout block To Workspace. The subsystem is activated by the “Trigger Signal”. When a trigger events occurs, the subsystem instantaneously updates its input value In1 and writes its output Out1. The basic blocks Sine Wave, Trigger Signal and To Workspace have the same sample time $T_S = 25$. The content blocks of the Triggered Subsystem have inherited sample time.

The subsystem is triggered at the rising edge of the square wave trigger control signal as shown in Figure 3.15. That corresponds to the time units 50, 125,... All blocks outside the Subsystem are executed on the same rate, equal to 25 units of time.

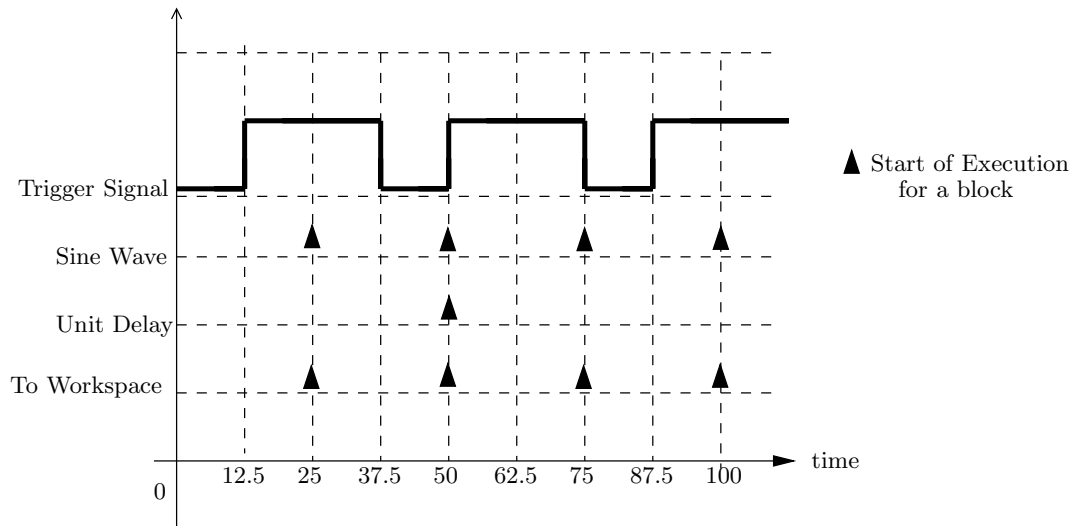


Figure 3.15: Execution time for blocks of the model 3.14

Enabled Subsystems

Enabled subsystems are time dependent. All the sample times are observed on a unique global time defined for the model, that means, execution is synchronized with respect to a global time.

The execution of enabled subsystems is constrained by the actual value of an external signal. That is, the subsystem (i.e. its inner blocks) executes only if the enabling signal has a positive value and stays unchanged otherwise.

Example 18 Figure 3.16 shows a Simulink example which contains an enabled Subsystem. Apart from the subsystem, it contains several atomic blocks such as the Sine Wave, the Pulse Generator, the Block A, the Block B and the Signal E that triggers the subsystem. The enabled subsystem contains the block C, the block D, inports and outputs. The block C and block D inside the subsystem have sample times $T_S=12.5$ and $T_S=25$, respectively.

All blocks outside the “Subsystem” execute independently of the enabling signal E. When the “Signal E” becomes positive, the block C and the block D execute at their assigned sample rates until the Signal E becomes again zero. Figure 3.17 shows the moment of execution for each block of the model.

3.4 Discussion

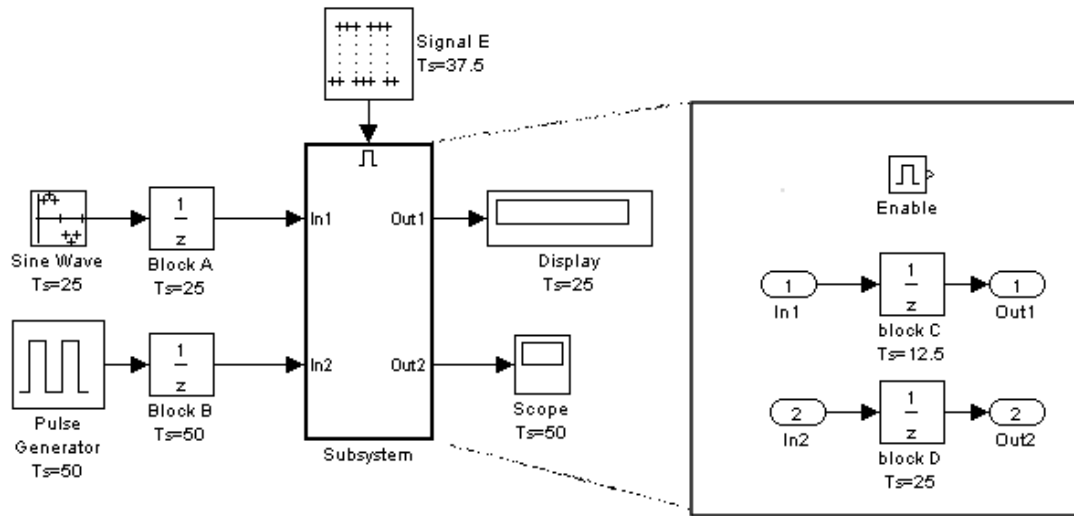


Figure 3.16: A Simulink model which contains an enabled subsystem

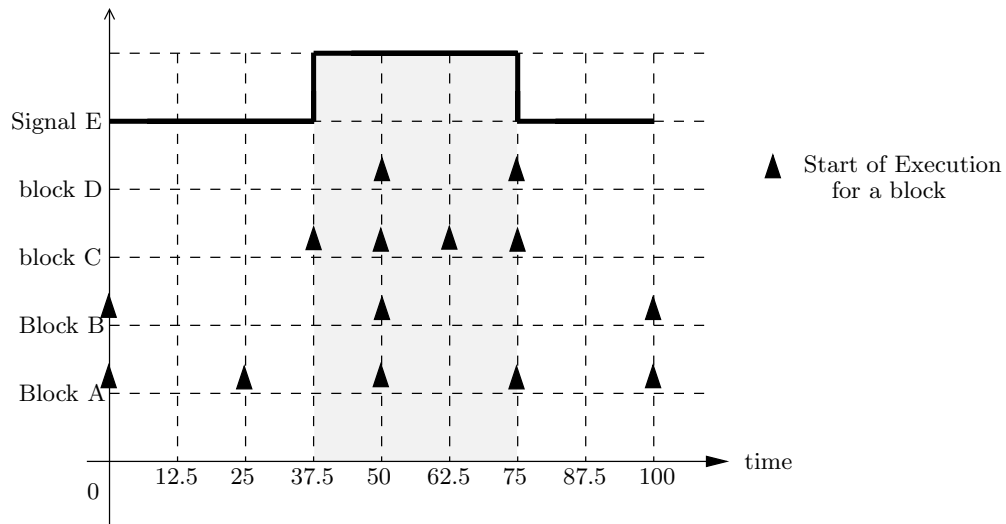


Figure 3.17: Execution time for blocks of the model 3.16

In this chapter we presented the LUSTRE language and the MATLAB/Simulink framework. *LUSTRE* is a synchronous languages with formal semantics developed at the VERIMAG laboratory for the past 25 years. On the top of the language, there is a number of tools, like code generator, model checker, tool for simulation of the system on design, etc., that constitute the LUSTRE platform. LUSTRE has been industrialized by Esterel Technologies in the SCADE tool. SCADE has been used from several companies in the area of aeronautics and automotive. It has been recently used for the development of the latest project of Airbus, the A380 carrier airplane.

MATLAB/Simulink is a commercial tool by MathWorks for modeling, simulating and analyzing dynamic systems. It offers a graphical interface and a wide variety of libraries that allow designer to model, simulate and measure performances of its system. Simulink is widely used in control theory, digital signal processing and Model-Based Design. Coupled with Real-Time Workshop by MathWorks, Simulink can automatically generate C code for given target execution platforms. Simulink has multitude of semantics which depend on user options for simulations.

Later in this document, we will show transformations of LUSTRE and MATLAB/Simulink to a component based framework dedicated for modeling synchronous systems. We demonstrate the transformations in several examples from LUSTRE and Simulink respectively.

Chapter 4

Modeling synchronous data-flow systems in BIP

In this chapter we present *Synchronous BIP*, an extension of the BIP framework [12] for modeling synchronous data-flow systems. Synchronous BIP components differ from general BIP components in two aspects. First, all atomic components are strongly synchronized under a global synchronization. This synchronization denotes the termination of a synchronous step during which all components perform some computation. Second, the behavior of atomic components is specified by a *subclass* of priority Petri nets.

Figure 4.1 shows three Synchronous BIP components A , B and C which are strongly synchronized through the global synchronization g_{sync} .

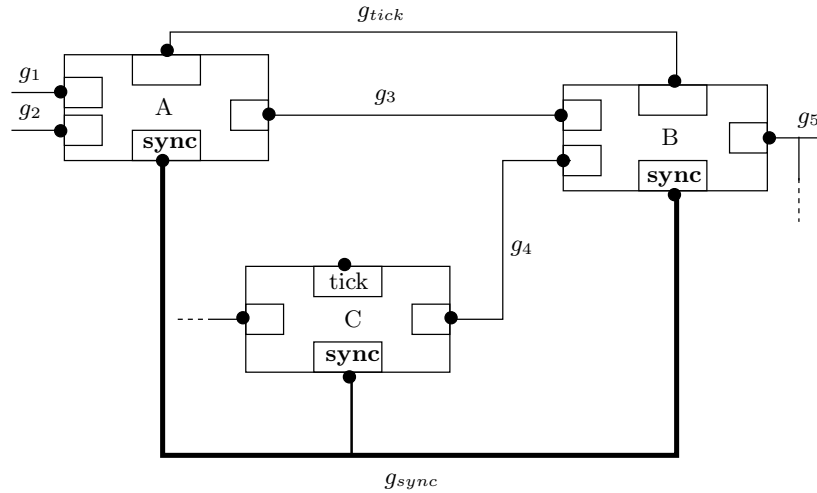


Figure 4.1: A synchronous system described in terms of Synchronous BIP

We consider two models for describing the behavior of atomic components in Synchronous BIP:

- *cyclic components*: They are based on priority Petri nets extended with an implicit “sync” transition which denotes termination of a synchronous step.
- *synchronous components*: They are described by modal flow graphs that is, directed acyclic graphs which represent implicitly the control flow for computation within a synchronous step.

Cyclic components resulted from the direct translation of synchronous systems into general BIP components. This model proved to be inappropriate since important safety properties like confluence and deadlock-freedom could not be guaranteed. Synchronous components were introduced in an attempt to describe the behavior of synchronous systems in BIP in such a way that these safety properties can be proved at low cost.

This chapter is structured as follows. Cyclic BIP components and their composition are presented in section 4.1. Section 4.2 presents the Synchronous BIP components defined using modal flow graphs. In these two sections, we define the semantics for both cyclic BIP components and synchronous BIP components.

Section 4.3 provides sufficient conditions for deadlock-freedom and confluence. The Synchronous BIP language is presented in section 4.4 and illustrated with examples. Section 4.5 presents related work. We conclude and we present the main applications of Synchronous BIP for modeling of synchronous systems in section 4.6.

4.1 Cyclic BIP Components

In this section we provide an initial model for describing synchronous systems in BIP, the *cyclic BIP components*. The behavior of a *cyclic component* is described by a priority safe-Petri net extended with an implicit *sync* transition which denotes the termination of a step.

4.1.1 Modeling Cyclic BIP Atomic Components

The transitions of this Petri-net are labeled with elements of a set of ports P and a *priority order*, a strict partial order $\prec \subset P \times P$. Furthermore, transitions may be labeled with guards and functions representing data transformations. The Petri net has a set of initial and a set of final places. When only no non-final places are marked, a step can terminate by executing the specific transition labeled by *sync*. The *sync* transition is executed synchronously by all components. Termination of a step consists in removing the tokens from final places and putting a token in each initial place. Implicitly, the priority order requires that *sync* has lower priority than any other port to ensure maximal computation in a step. The formal definition for an atomic cyclic BIP component is given below:

Definition 12 (Cyclic BIP Component: Syntax) *A cyclic BIP component B is a tuple (X, P, N, \preceq) where:*

- X is a set of data variables
- P is a set of ports p , each one labelled with a subset of variables $X_p \subseteq X$, the ones exported on interactions through p .
- $N = (L, T, F, L_0, L_f)$ is an extended 1-safe Petri net:
 - L is a finite set of places;
 - T is a finite set of transitions τ labelled by $(p_\tau, gu_\tau, f_\tau)$ where:
 - * p_τ is the port triggered by the transition τ ,
 - * gu_τ is the guard of τ , that is a predicate on X and
 - * f_τ is the update function associated with the transition τ . As already defined in Chapter 2, $f_\tau = (f_\tau^{(x)})_{x \in X}$, that is, for every $x \in X$, it provides an arbitrary expression on X defining the next (updated) value for x . We concretely represent f_τ as sequential programs operating on data X .

- $F \subseteq L \times T \cup T \times L$ is the token flow relation,
- $L_0 \subseteq L$ is the set of initial places,
- $L_f \subseteq L$ is the set of final places,
- $\prec \subseteq P \times P$ is a priority order on ports, that is a strict partial order on the set of ports.

Note that the set of initial and final places can intersect, that is $L_f \cap L_0 \neq \emptyset$.

Example 19 Figure 4.2 shows a cyclic BIP component that produces a tock every P ticks. Initial places are marked with a token; final places are grayed. At every step, it executes the tick transition and then, during the same step, it increases the local variable x by executing the update transition. Whenever x reaches the value P , the component can also execute the tock transition and reset x to 0. In this situation, the tock and update transitions are conflicting, however, the associated priorities enforce the execution of tock before update if both transitions are possible.

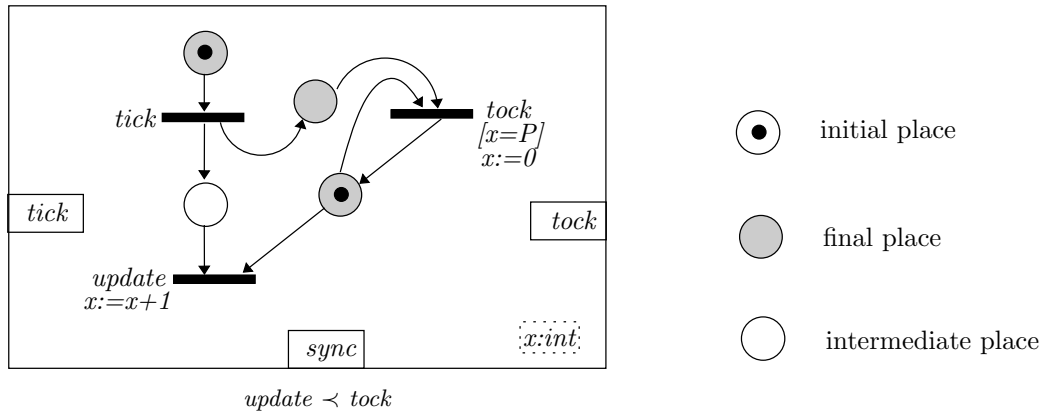


Figure 4.2: The tick-tock cyclic BIP component

Definition 13 (Cyclic BIP Component: Semantics) The operational semantics of a cyclic BIP component $B = (X, P, N, \prec)$ with $N = (L, T, F, L_0, L_f)$ is defined as the labelled transition system $S = (Q, \Sigma, \rightarrow)$ where

- $Q = \mathcal{M} \times \mathcal{D}^X$ is the set of states where $\mathcal{M} = \{m : L \rightarrow \mathbb{N}\}$ is the set of 1-safe markings and $\mathcal{D}^X = \{v : X \rightarrow \mathcal{D}\}$ is the set of valuations of variables,
- $\Sigma = \{(p, v, v') \mid p \in P, v \in \mathcal{D}^{X_p}, v' \in \mathcal{D}^{X_p}\} \cup \{sync\}$ is the set of labels. A label (p, v, v') , as already defined in Chapter 2, marks instantaneous data change through the port p . The current valuation v is sent and a new valuation v' is received for the set of variables X_p . We note a label (p, v, v') as $p(v/v')$.
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions defined by the rules below:

Rule 1:

	control	data	
$\tau \in T$	$m \in \mathcal{M}, m' \in \mathcal{M}$	$v \in \mathcal{D}^X, v' \in \mathcal{D}^X$	
labeled by	$\bullet \tau \leq m$	$gu_\tau(v) = \text{true}$	(read v) guard
$(p_\tau, gu_\tau, f_\tau)$	$m' = m - \bullet \tau + \tau \bullet$	$v' = f_\tau(v)$	(write v') action

$$(m, v) \xrightarrow{p_\tau(v/v')}_0 (m', v')$$

Rule 2:

control	data	guard
$m \leq m_{L_f}$		
$m' = m_{L_0}$	$v' = v$	action

$$(m, v) \xrightarrow{sync}_0 (m', v')$$

Rule 3:

$$\frac{(m, v) \xrightarrow{p_\tau(v/v')}_0 (m', v') \quad \neg(\exists p'. p_\tau \prec p' \wedge (m, v) \xrightarrow{p'}_0)}{(m, v) \xrightarrow{p_\tau(v/v')}_0 (m', v')}$$

Rule 4:

$$\frac{(m, v) \xrightarrow{sync}_0 (m', v') \quad \neg(\exists p_\tau. (m, v) \xrightarrow{p_\tau(v/v')}_0)}{(m, v) \xrightarrow{sync}_0 (m', v')}$$

Rule 1 and **Rule 2** define moves \rightarrow_0 of the behavior without priorities. **Rule 1** is the usual firing rule of transitions in Petri nets extended with global data. **Rule 2** defines *sync* transitions which denote the end of a step and the beginning of the next one. *sync* transitions can be executed whenever the current marking does not contain tokens in non-final places, and their effect is to restore the initial marking, while keeping the data unchanged. **Rule 3** and **Rule 4** define the moves \rightarrow of a synchronous component, by restricting \rightarrow_0 with respect to priorities. **Rule 3** is simply the application of the priority rule specified by the priority order \prec . **Rule 4** ensures that the *sync* transition is executed only if no other transition can.

Let us note that the rules **R1** and **R3** correspond to the standard BIP semantics, whereas rule **R2** defines the implicit *sync* step and **R4** gives the priority order in case of conflict between the *sync* and any other transition.

4.1.2 Composition of Cyclic BIP Atomic Components

We define composition parametrized by interactions as an operation of cyclic components. This operation is partial: the result of the composition is defined as a cyclic component only if the priority order associated to it is acyclic. An interaction is interpreted as in Definition 8 of Chapter 2.

Definition 14 (Composition of Cyclic BIP Components: Semantics) Let a set of cyclic components $\{B_i = (X_i, P_i, N_i, \prec_i)\}_{i=1,n}$ defined on disjoint sets of variables and ports. Let γ be a set of interactions on ports $\cup_{i=1}^n P_i$ such that each interaction uses at most one port of every component, that is for all $a \in \gamma$, for all $i \in \overline{1, n}$, $|a \cap P_i| \leq 1$. The composition $\gamma(B_1, \dots, B_n)$ is a partial operation defining the cyclic component $B = (X, P, N, \prec)$ where

- the set of variables is $X = \cup_{i=1}^n X_i$,

- the set of ports P is the set of interactions γ . Moreover, for each interaction $a \in \gamma$, and for X_p the set of exported variables for each port p , the set of its exported variables is $X_a = \cup_{p \in a} X_p$,
- the Petri net $N = (L, T, F, L_0, L_f)$ is obtained from the set of the Petri nets $\{N_i = (L_i, T_i, F_i, L_{0i}, L_{fi})\}_{i=1,n}$ as follows:
 - the set of places $L = \cup_{i=1}^n L_i$,
 - the set of transitions T corresponds to sets of interacting transitions

$$T = \left\{ \langle a, \{\tau_i\}_{i \in I} \rangle \left| \begin{array}{l} a \in \gamma, \ I \subseteq \overline{1, n} \text{ such that} \\ \forall i \in I. \tau_i \in T_i \wedge P_a = \{p_{\tau_i}\}_{i \in I} \end{array} \right. \right\}$$

Each transition τ is labeled by $(p_\tau, gu_\tau, f_\tau)$ where:

- * p_τ is the interaction a
- * $gu_\tau = \wedge_{i=1}^n gu_{\tau_i} \wedge G_a$ is the guard and it is a predicate on the set of variables X . gu_{τ_i} is the guard of the transition τ_i and G_a the guard of the interaction a .
- * $f_\tau = F_a; (\sqcup_{i=1}^n f_{\tau_i})$ is the data transfer function. It consists of the interaction function F_a , followed by local functions f_{τ_i} in arbitrary order (the order of computation is irrelevant as the data of the components are disjoint).
- the token flow relation F of the net is defined as

$$F = \left\{ (l, \langle a, \{\tau_i\}_{i \in I} \rangle) \mid (\exists j \in I, l \in \bullet \tau_j) \right\} \cup \left\{ (\langle a, \{\tau_i\}_{i \in I} \rangle, l) \mid (\exists j \in I, l \in \tau_j^\bullet) \right\}$$

- the set of initial places L_0 is $\cup_{i=1}^n L_{0i}$,
- the set of final places L_f is $\cup_{i=1}^n L_{fi}$,
- the relation \prec is the strict transitive closure of the relation \prec_0 defined as the extension of individual priority orders \prec_i to interactions: $a_1 \prec_0 a_2$ iff $\exists i \in \overline{1, n}. \exists p_{i1} \in P_{a1} \cap P_i. \exists p_{i2} \in P_{a2} \cap P_i$ such that $p_{i1} \prec_i p_{i2}$. The composition is defined only if this relation is a strict partial order.

Example 20 *Composition of two cyclic components is illustrated in Figure 4.3. Two tick-tock components are composed through the interactions γ^{toTi} and γ^{sync} . The interaction γ^{toTi} synchronizes the tick of the left component and the tick₂ of the right one. The γ^{sync} interaction synchronizes strongly the sync ports of both components.*

The resulting component is shown in Figure 4.4. The transition $tock_1 tock_2$ corresponds to the interaction γ^{toTi} . Guards and update functions are inherited from the atomic components. The composed component produces a $tock_2$ every $P_1 \times P_2$ ticks.

An essential property of synchronous systems is termination of steps, in particular steps must be deadlock-free. Another requirement is confluence of computation within a step which means that the overall behavior is deterministic when system states are observed only at the end of each step. For some synchronous languages e.g. Lustre, these properties can be ensured by checking very simple sufficient conditions [43]. Proving these properties using automata and standard Petri nets is hard. For that, we define the class of *modal flow components* where priority Petri nets are replaced by *modal flow graphs*. These graphs correspond to a subclass of priority Petri nets where arbitrary control flow is restricted to an acyclic graph of dependencies between ports. For modal flow graphs, deadlock-freedom and confluence can be decided at

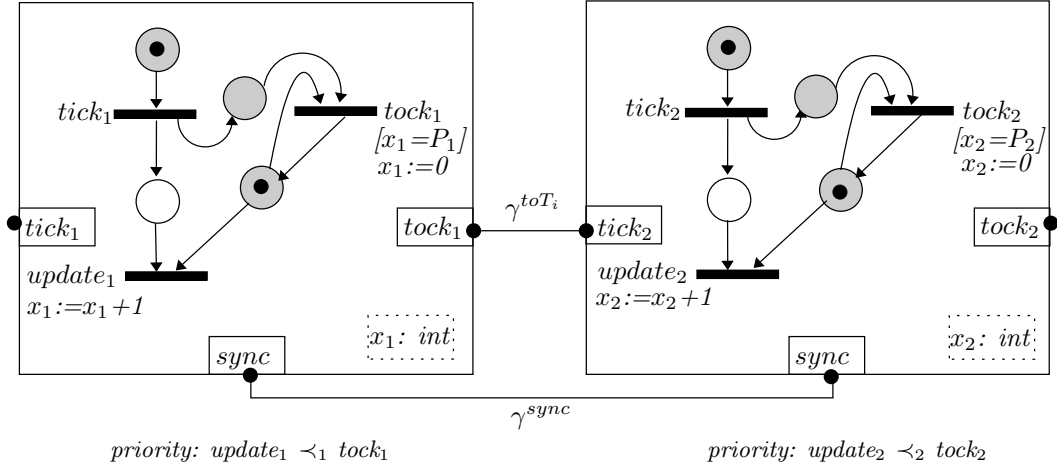


Figure 4.3: Example of composition of cyclic BIP components

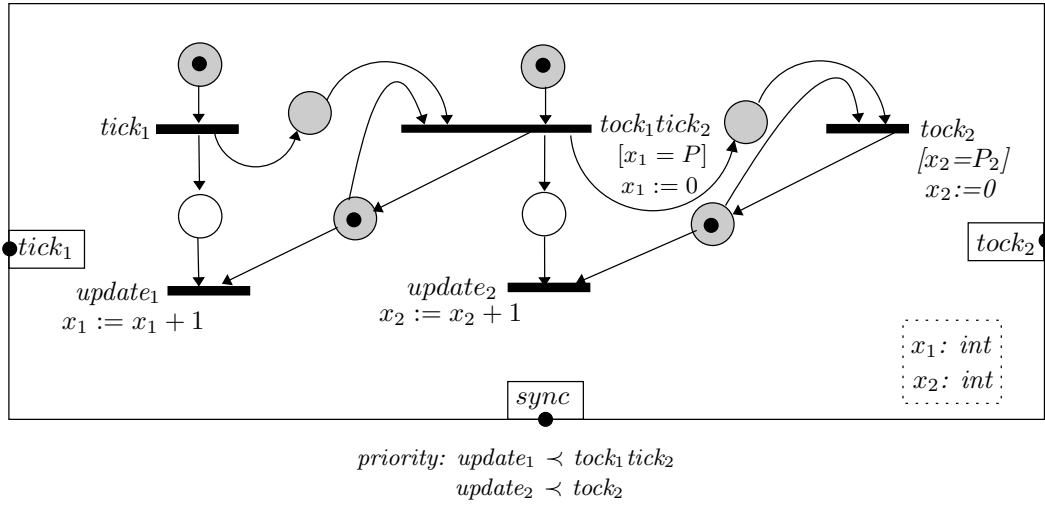


Figure 4.4: The resulted components from the composition of the cyclic BIP components of Figure 4.3

low cost. The causality relation between events is expressed explicitly through three types of dependencies. These dependencies exhibit a part of all possible combinations between a pair of events. These are the sufficient combinations needed for describing the behavior of some synchronous formalisms like Lustre.

4.2 Synchronous BIP Components

A Synchronous BIP model consists of synchronous BIP components which are strongly synchronized through implicit synchronizations. The behavior of atomic components is described as *modal flow graphs*.

For a given set of ports P , a modal flow graph is a directed acyclic graph with nodes P and edges representing the union of three binary relations. Each relation expresses a different kind of causal dependency (modality) between pairs of ports p and q within a step:

- q *strongly depends* on p if the execution of p *must* be followed by the execution of q . That is, p and q cannot be executed independently, only the sequence pq is possible.
- q *weakly depends* on p if the execution of p *may* be followed by q . That is either p can be executed alone or the sequence pq .
- q *conditionally depends* on p if when both p and q are executed, then q must follow p . Conditional dependency requires that if p and q occur together, then only the sequence pq is possible; otherwise p or q may be independently executed.

Figure 4.5 illustrates the graphical notation used for the three dependencies as well as their possible executions in a synchronous step.

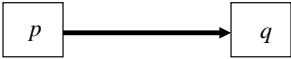
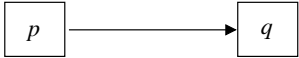
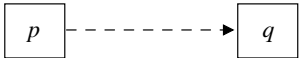
Dependency	Graphical Representation	Interpretation	Execution
<i>strong</i>		q must follow p	pq
<i>weak</i>		q may follow p	p, pq
<i>conditional</i>		q never precedes p	p, q, pq

Figure 4.5: The three causal dependencies and the possible executions in a synchronous step

4.2.1 Modeling Synchronous BIP Atomic Components

A BIP component that describes the behavior of a synchronous system in terms of modal flow graphs is called *Synchronous BIP components*. A formal definition for a Synchronous BIP component is given below.

Definition 15 (Synchronous BIP Component: Syntax) A Synchronous BIP component B^f is defined as a tuple (X, P, D) :

- X is a set of data variables,
- P is a set of ports p , each one being associated with a triple (X_p, gu_p, f_p) where
 - $X_p \subseteq X$, the set of variables exported through p ,
 - gu_p , the triggering condition of p , that is a predicate defined on X ,
 - f_p , an update function, that is a state transformer function on X . As already mentioned, $f_p = (f_p^{(x)})_{x \in X}$, that is, for every $x \in X$, it provides an arbitrary expression on X defining the next (updated) value for x . We concretely represent f_p as sequential programs operating on data X .
- $D = (D_s, D_w, D_c)$ is a triple of causal dependency relations between ports. The relations $D_s, D_w, D_c \subseteq P \times P$ denote respectively strong, weak and conditional dependency and are such that their union $D_s \cup D_w \cup D_c$ is acyclic.

Example 21 Figure 4.6 represents as a Synchronous BIP component, the tick-tock cyclic BIP component of Figure 4.2.

The port tock is weakly dependent on the tick port. Also, the update port is strongly dependent on tick and conditionally dependent on tock. Each time that the port update is executed, the value of x is increased by one. The only possible executions within a step are therefore (tick update) or (tick tock update) whenever x reaches P .

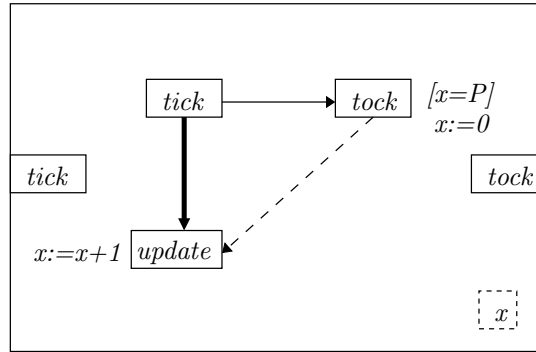


Figure 4.6: Tick-tock Synchronous BIP component

We use the following notation. For fixed $x = s, w, c$, we write $p \overset{x}{\rightsquigarrow} q$ to denote $(p, q) \in D_x$. We write $\overset{x}{\rightsquigarrow}^*$ to denote the reflexive and transitive closure of $\overset{x}{\rightsquigarrow}$. We write $p \rightsquigarrow q$ to denote $(p, q) \in D_s \cup D_w \cup D_c$ and \rightsquigarrow^* for its reflexive and transitive closure. Two ports p and q are called independent (noted $p \# q$) iff neither $p \rightsquigarrow^* q$ nor $q \rightsquigarrow^* p$.

For fixed $x = s, w, c$, we denote by $\min_x P$ the set of minimal ports with respect to D_x , that is $\min_x P = \{q \mid \neg \exists p. p \overset{x}{\rightsquigarrow} q\}$. We write $\min P$ to denote the set of minimal ports with respect to $D_s \cup D_w \cup D_c$, that is $\min P = \{q \mid \neg \exists p. p \rightsquigarrow q\}$.

4.2.2 Well-triggered Modal Flow Graphs

We introduce now the notion of *well-triggered* modal flow graphs. This notion ensures consistency, between the three types of dependencies, defined by the following constraints:

1. each port p has a unique minimal strong cause

$$|\{q \in \min_s P \mid q \overset{s}{\rightsquigarrow}^* p\}| = 1$$

2. each port p has exclusively either strong or weak causes.

In a well-triggered modal flow graph, for a port p , we denote its minimal strong cause by $\text{root}(p)$.

By describing well-triggered modal flow graphs, we provide syntactical restrictions that exclude deadlock situations that may occur within a step. Some examples are illustrated in Figure 4.8. For Figure 4.8(a), if the strong cause q_1 is executed but not the strong cause q_2 , then we will reach a situation where the execution at p is disabled and the component cannot progress to reach the *sync*. Similar is the situation in Figure 4.8(b), where the graph reaches a deadlock situation if port q_1 is executed but not q_2 and consequently p stays disabled. However, the graph of Figure 4.8(c) will never reach a deadlock situation since a conditional dependency never blocks the execution. If port q_2 is executed but not q_1 , the conditional dependency allows port p to be executed.

For the sequel, we consider only well-triggered modal flow graphs. As we will show in Section 4.3, for this type of modal flow graphs we provide simple conditions that ensure deadlock-freedom and confluence. Moreover, for some synchronous languages like LUSTRE (see Chapter 5), well-triggered modal flow graphs are enough to describe their operators.

Well-triggered modal flow graphs can be decomposed as shown in Figure 4.7.

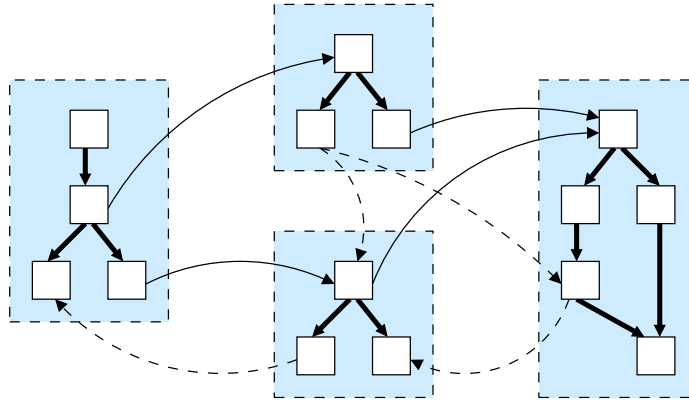


Figure 4.7: Well-triggered components

The strong dependency relation defines a set of connected subgraphs involving all the ports of the component. Each one of these subgraphs has a single root which is the common cause for its ports. Weak dependencies express triggering of the root of a subgraph by some port of another subgraph. Finally, conditional dependencies may relate ports of different subgraphs provided the acyclicity property is not violated.

We define the semantics of synchronous components which are well-triggered in terms of cyclic BIP components.

Definition 16 (Synchronous BIP Component: Semantics) The operational semantics of a well-triggered synchronous component $B^f = (X, P, D)$ is a cyclic component $B = (X, P, N, \prec)$:

- the set of variables is X ,
- the set of ports is P ; moreover, for each port p the associated set of exported variables is X_p ,

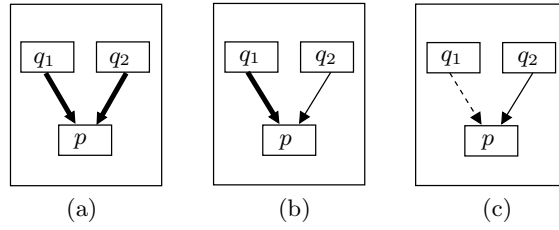


Figure 4.8: Examples of modal flow graphs, non well-triggered (a) and (b) and well-triggered (c)

- the Petri net $N = (L, T, F, L_0, L_f)$ is defined by:
 - the set of places L is isomorphic to the set $D_s \cup D_w \cup D_c$ augmented with the set of minimal ports. That is $L = \{l_{p,q}^x \mid p \overset{x}{\rightsquigarrow} q\} \cup \{l_p \mid p \in \min P\}$,
 - the set of transitions T is isomorphic to the set of ports P , that is $T = \{\tau_p \mid p \in P\}$. Moreover, for any transition τ_p we define the triple $(p_\tau, g_{\tau_p}, f_{\tau_p})$, where:
 - * p_τ is the associated port
 - * g_{τ_p} is the guard of τ_p and it is a predicate on X
 - * f_p is the update function. As already defined in Definition 15, $f_{\tau_p} = (f_{\tau_p}^{(x)})_{x \in X}$, that is, for every $x \in X$, it provides an arbitrary expression on X defining the next (updated) value for x . We concretely represent f_{τ_p} as sequential programs operating on data X ,
 - the token flow relation $F \subseteq L \times T \cup T \times L$, is constructed as follows:
 - * for each $p \in \min P$ add (l_p, τ_p) to F ,
 - * for each dependency $p \overset{x}{\rightsquigarrow} q$ add $(\tau_p, l_{p,q}^x), (l_{p,q}^x, \tau_q)$ to F ,
 - * for each conditional dependency $p \overset{c}{\rightsquigarrow} q$ add $(l_{p,q}^c, \tau_{root(p)})$ to F . This relation implies that execution of q disables further execution of the $root(p)$, in the same step,
 - the set of initial places L_0 corresponds to minimal ports and conditional dependencies that is $L_0 = \{l_p \mid p \in \min P\} \cup \{l_{p,q}^c \mid p \overset{c}{\rightsquigarrow} q\}$,
 - the set of final places L_f consists of all places corresponding to all but strong dependencies $L_f = L \setminus \{l_{p,q}^s \mid p \overset{s}{\rightsquigarrow} q\}$.
- the priority order $\prec = (\rightsquigarrow^*)^{-1} \setminus Id$, that is $p \prec q$ iff $q \rightsquigarrow^* p$ and $q \neq p$, for all $p, q \in P$.

The Petri nets representing Synchronous BIP components satisfy the following trivial properties: 1) every place has at most one incoming transition, 2) every place $l_{p,q}^c$ corresponding to a conditional dependency belongs to a cycle, 3) initially, there is precisely one token in every cycle of the net.

The mapping of modal flow graphs to Petri nets is done by associating with each port a transition of the Petri net and the dependencies transformed as shown in Figure 4.9. These Petri nets represent valid execution for one synchronous step. Termination of a step consists in removing the tokens from final places and putting a token in each initial place. Guards and update functions are inherited from ports to the corresponding transitions.

Notice that the above construction rules of the Petri net enforce the three kinds of dependencies between ports. Strong and weak dependencies are obviously enforced by the net. An

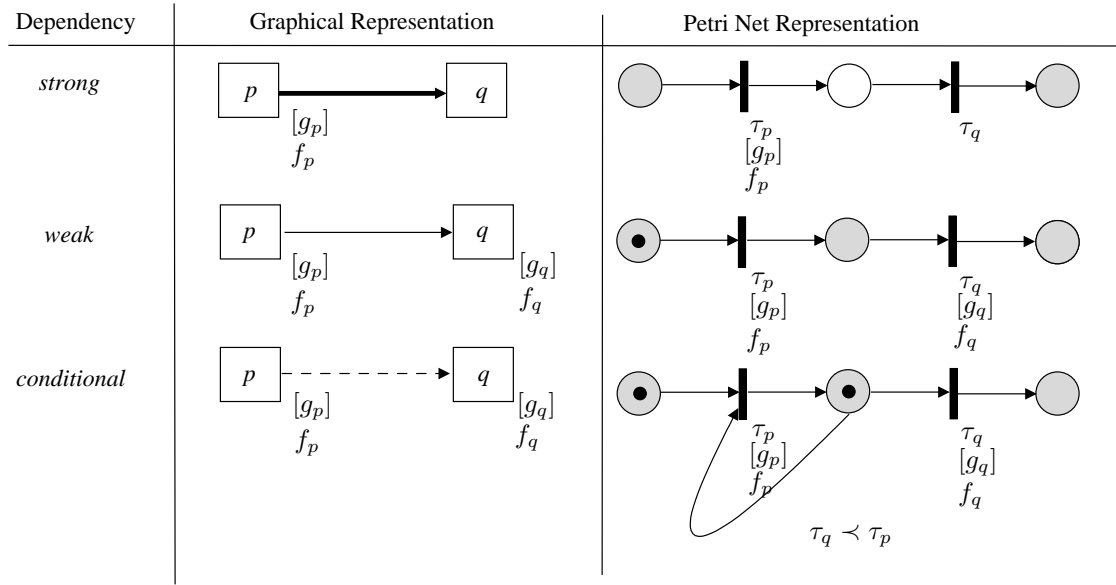


Figure 4.9: The correspondence of causal dependencies in Petri nets

initial empty place $l_{p,q}$ between τ_p and τ_q will prevent the execution of τ_q before τ_p . Moreover, if the place is not final, the execution of τ_p will require the execution of τ_q before the end of the step. Concerning conditional dependencies $p \rightsquigarrow q$, the Petri net ensures that the execution of τ_q disables any further execution of $\tau_{root(p)}$ and consequently of τ_p . For the conditional dependency of Figure 4.9, $root(p) = p$.

Example 22 The tick-tock synchronous component shown in Figure 4.10 (right), is well-triggered. Its semantics is defined by the tick-tock cyclic component in Figure 4.10 (left). As explained in Example 19), due to initial marking, in one step, transition *tick* can be executed followed by the execution of the transition *update*, increasing x by one. Whenever x reaches P , both transitions *update* and *tock* can be executed. This conflict is resolved and in the same time, maximal computation is ensured using the priority $update \prec tock$ that enforces execution of *tock* before *update*.

Notice that guards and update functions in ports of the synchronous components are inherited from the transitions of the cyclic component.

The next result gathers some additional properties.

Proposition 1 Priority Petri nets representing modal flow graphs meet the following properties

1. Every reachable marking has at most one token in every cycle of the net.
2. Each transition is executed at most once in every step.
3. Are 1-safe.

Proof.

1. This property is an inductive invariant on the set of reachable markings and holds because each place in a cycle has a unique incoming transition – that is any attempt to put a token into a cycle will first remove a token from the same cycle.

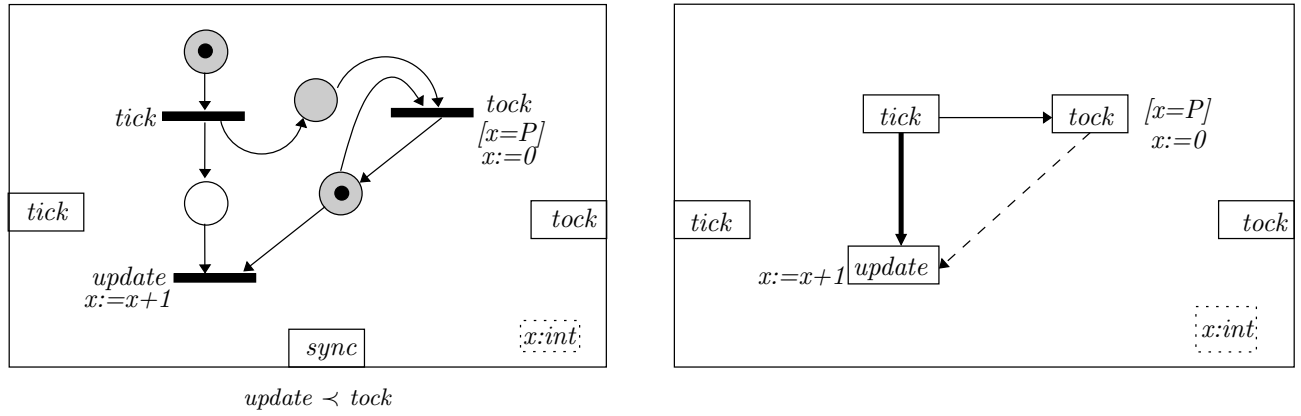


Figure 4.10: The Tick-tock synchronous component (right) and its semantics (left)

2. Minimal ports p can only be executed at most once since they will remove the token in the corresponding initial place l_p . For any other port p , we consider that all his direct preceding ports q are executed at most once. If q is related to p through a strong or weak dependency, by construction, p is executed at most once. If q is related to p through a conditional dependency, the place $l_{q,p}$ has initially a token and belongs to a cycle of the net. By executing p , the cycle containing $l_{q,p}$ becomes empty and will remain empty in any further execution of the net;
3. Given the previous result, it follows that every place will receive a token at most once. So every place may have at most one token, if it is initially empty, or two tokens, if it contains initially a token. However, places that contain initially tokens belong to cycles and cycles contain at most one token globally. So the net is 1-safe.

□

4.2.3 Composition of Synchronous BIP Atomic Components

We lift composition of cyclic BIP components to Synchronous BIP components, as follows.

Definition 17 (Composition of Synchronous BIP Components: Semantics) Let $\{B_i^f = (X_i, P_i, D_i)\}_{i=1,n}$ be a set of synchronous components defined on disjoint sets of variables and ports. That is, for variables X_i, X_j and ports P_i, P_j it holds $X_i \cap X_j = \emptyset$ and $P_i \cap P_j = \emptyset$. Let γ be a set of interactions on ports $\cup_{i=1}^n P_i$ such that

- each interaction uses at most one port of every component, that is for all $a \in \gamma$, for all $i \in \overline{1, n}$ $|a \cap P_i| \leq 1$,
- each port belongs to at most one interaction, that is for all $p \in \cup_{i=1}^n P_i$ $|\{a \mid p \in P_a\}| \leq 1$,

We define the composition $\gamma(B_1^f, \dots, B_n^f)$ as the modal flow component $B^f = (X, P, D)$ where

- the set of variables X is $\cup_{i=1}^n X_i$,
- the set of ports P is the set of interactions γ . Moreover, for every interaction a of γ , we define the tuple (X_a, g_a, f_a) , where:
 - X_a is the set of exported variables such that $X_a = \cup_{p \in P_a} X_p$ and X_p the set of variables exported at the port p ,

- g_a is the guard such that $g_a = (\bigwedge_{p \in P_a} g_p) \wedge G_a$, a predicate on X
- f_a is the data transfer function such that $f_a = F_a; (\sqcup_{p \in P_a} f_p)$. where \sqcup defines parallel composition. Figure 4.11 depicts the composition of ports with guards and transfer functions.

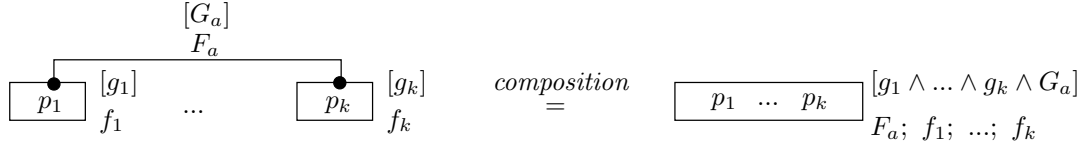


Figure 4.11: Composition of ports

- the set of dependencies $D = (D_s, D_w, D_c)$ are inherited from atomic components, that is for every $x = s, w, c$ we have $D_x = \{(a_1, a_2) \mid \exists i \in \overline{1, n}. \exists p_1 \in P_{a_1} \cap P_i, p_2 \in P_{a_2} \cap P_i \text{ such that } (p_1, p_2) \in D_{xi}\}$

Notice that composition amounts to merging nodes belonging to the same interaction without changing the dependency relations. Composition is a partial operation because, its result is a valid synchronous component only if the set of derived dependencies is acyclic.

Example 23 The composition of two Synchronous BIP components is illustrated in Figure 4.12.

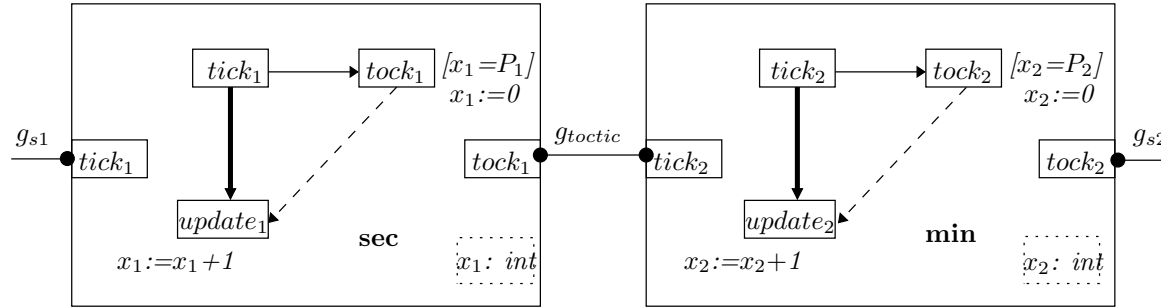


Figure 4.12: Example of composition of Synchronous BIP atomic components

Two tick-tock components *sec* and *min* respectively are composed by synchronizing the tock of the first component and the tick of the second one through the control flow interaction γ^{toTi} .

Figure 4.13 illustrates the produced component from the composition of the two atomic components *sec* and *min*. The resulting component produces a tock₂ every P_2 tock₁.

Let us observe that the result from the composition of Synchronous BIP components is not the same operation as composition of cyclic BIP components. These differ because conditional dependencies do not have a local interpretation e.g. $p \xrightarrow{c} q$ implies that execution of transition q disables further execution $root(p)$. But, the minimal strong cause $root(p)$ of p can denote different actions within the Synchronous BIP component and the composed Synchronous BIP component.

Example 24 This example validates the previous observation. Figure 4.14 (a) shows two Synchronous BIP components strongly synchronized through the interaction γ . The interaction connects the ports q and r and it is associated with the data-transfer function $y := x$.

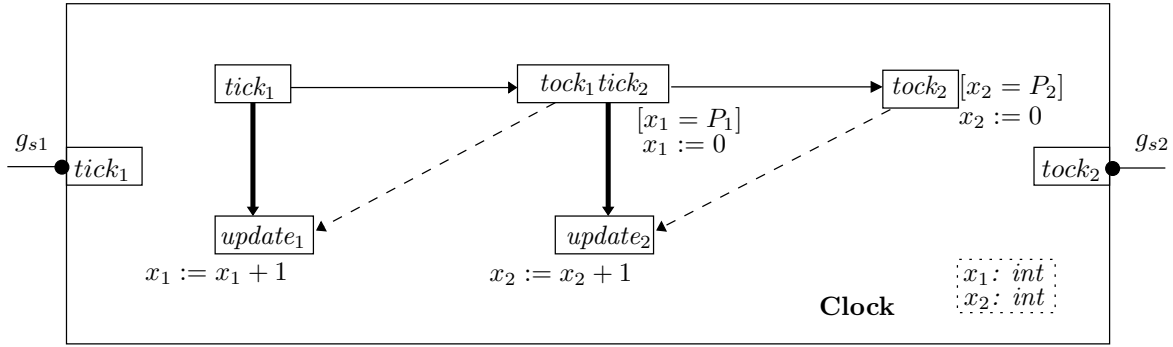


Figure 4.13: The produced component from the synchronization of the composition of atomic components shown in figure 4.12

The resulted composed component is shown in Figure 4.14 (b). Ports q and r are composed to a single port which executes the associated functions each time this port is triggered. The behavior of the execution of the composed component is illustrated on the following table:

x :	0	1	2	3	4	5	6	7	...
y :	—	0	2	4	6	8	10	12	...

This component is transformed to cyclic BIP component as shown in Figure 4.14 (c). For components of Figures 4.14 (b), (c), the possible executions are in one step either s or p , qr , s .

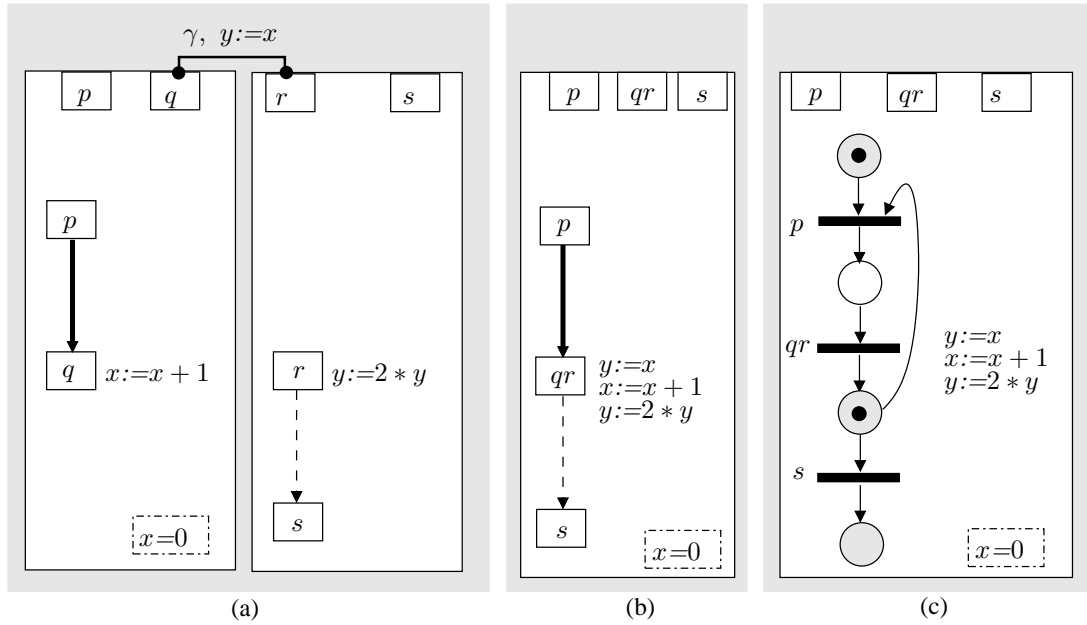


Figure 4.14: (a) Two Synchronous BIP components strongly synchronized through the interaction γ , (b) the component obtained by the composition of the two Synchronous BIP components, (c) the cyclic BIP component corresponding to the composed Synchronous BIP component

Figure 4.15 (a) is the translation of Figure 4.14 (a) into cyclic BIP components. Figure 4.15 (b) shows the resulted cyclic BIP component obtained by the composition of the two components

of Figure 4.15 (a). A simplification of this component is shown in Figure 4.15 (c). The possible executions are either p , qr , s or s , p , qr .

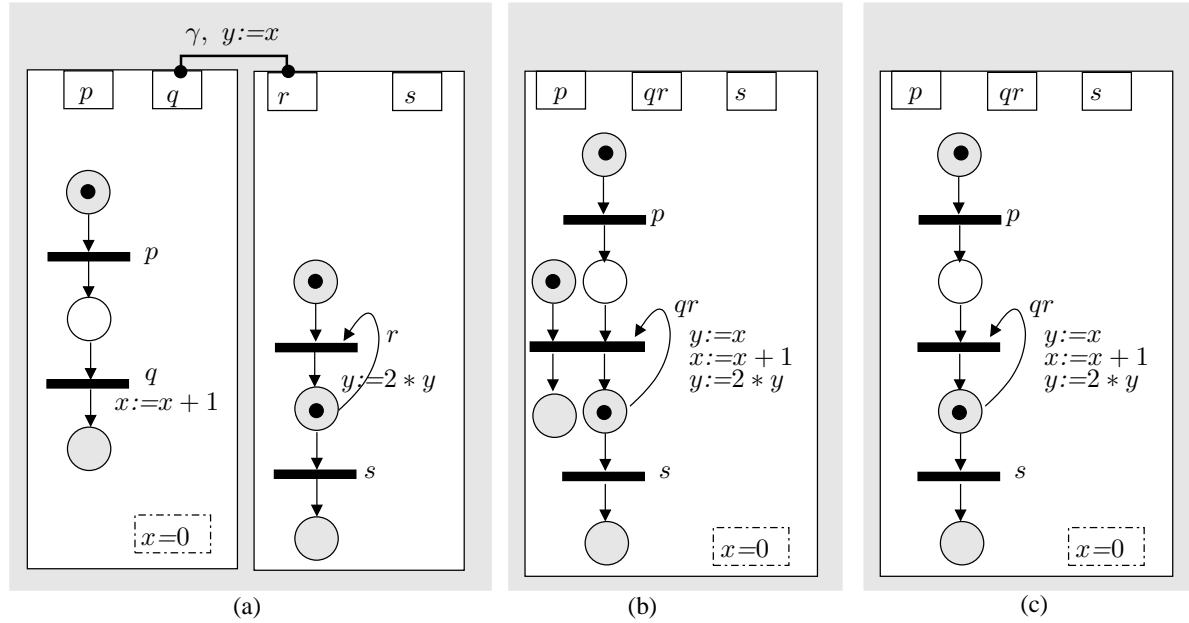


Figure 4.15: (a) The cyclic BIP components corresponding to the Synchronous BIP components of Figure 4.14 strongly synchronized through the interaction γ , (b) the Synchronous BIP compound component obtained by the composition of the two cyclic BIP components, (c) a simplified form for the Petri net of the compound component

We observe that the executions of the composed cyclic BIP component and the Synchronous BIP component are not identical. The cyclic BIP component produces the execution s , p , qr which does not correspond to the desirable behavior. If both s and r appear in the same step then s must be executed before r .

4.3 Structural Properties of Synchronous BIP Components

In this section we will give the main results of the Synchronous BIP components. Synchronous BIP components can be proved deadlock-free and confluent (i.e., deterministic) by checking syntactic conditions. These conditions are formally stated by theorems below. We start by giving concrete definitions about deadlock-freedom and confluence.

Definition 18 A modal flow graph is deadlock-free if all tokens have been removed from non-final places or equivalently when the set $P_0 = \{q \mid \exists p \cdot p \xrightarrow{s} q \text{ and } l_{p,q}^s \text{ has a token}\}$ is empty.

Theorem 1 (Deadlock-freedom) A well-triggered Synchronous BIP component $B^f = (X, P, D)$ is deadlock-free if every port p with strong causes has its guard true that means $g_p = \text{true}$.

Proof. First situation, if no strong dependencies exist, all places in the Petri net are final therefore the *sync* transition is always enabled. Consequently, no deadlock is possible.

Second situation, if there exist strong dependencies, a deadlock potentially occurs only when the *sync* transition is not enabled. This happens only when there are non-final places containing

tokens or equivalently when the set

$P_0 = \{q \mid \exists p \cdot p \xrightarrow{s} q \text{ and } l_{p,q}^s \text{ has a token}\}$ is not empty. Let now define the set P_1 to be

$P_1 = \{r \mid \exists q \in P_0 \cdot q \xrightarrow{s}^* r\}$ the set of ports that are transitively strongly dependent on ports in P_0 . Obviously, we have $P_0 \subseteq P_1$. Intuitively, the set P_1 contains all the ports which have strong dependencies and which remain to be executed in the current step.

Choose $r_0 \in P_1$, an arbitrary minimal element of P_1 . We will show that r_0 is enabled and therefore, there is no deadlock. First, r_0 is a port with strong dependencies. From the hypothesis of the theorem, we know that its guard is true, therefore whether it is enabled or not depends only on the control (i.e., the markings of the net) and not on data. By contradiction, assume that there are missing tokens in one of the previous places l_{u,r_0}^x of r_0 . Since the component is well-triggered, we distinguish two cases, depending on the dependency x :

1. $x = s$, the place l_{u,r_0}^s comes from a strong dependency. First, if there is no token in l_{u,r_0}^s it follows that u has not been executed within a step. Second, since the component is well-triggered, we have $root(u) = root(r_0)$. Third, $r_0 \in P_1$ implies that $root(r_0)$ has been already executed within the step. Consequently, we have also $u \in P_1$, that is u remains to be executed in the current step. Since we have $u \xrightarrow{s}^* r_0$, this contradicts the minimality of r_0 in P_1 .
2. $x = c$, the place l_{u,r_0}^c comes from a conditional dependency. If there is no token in l_{u,r_0}^c , it means that the token has been consumed for the firing of $root(u)$ and not yet produced by u . Hence u belongs to the set P_1 and we have $u \xrightarrow{c}^* r_0$. Again, this contradicts the minimality of r_0 in P_1 .

□

A modal flow graph is confluent if it has deterministic behavior. That is, it always has the same output independently of the order of execution of conflicted ports. A formal definition for a *confluent* modal flow graph is given below:

Definition 19 A modal flow graph is confluent iff for each state (m, v) such that $(m, v) \rightarrow^* (m_1, v_1)$ and $(m, v) \rightarrow^* (m_2, v_2)$ then $(m_1, v_1) = (m_2, v_2)$.

Theorem 2 (Confluence) A well-triggered Synchronous BIP component $B^f = (X, P, D)$ is confluent if for every pair of independent ports $p_1 \# p_2$, their associated guarded actions are independent, that is:

- $X_{p_1} \cap X_{p_2} = \emptyset$
- $use(g_{p_1}) \cap (X_{p_2} \cup def(f_{p_2})) = \emptyset$
- $use(g_{p_2}) \cap (X_{p_1} \cup def(f_{p_1})) = \emptyset$

Proof. Whenever there is a choice between executing two ports p_1 and p_2 after applying priorities, it follows that p_1 and p_2 are independent. That is, by definition, priorities select enabled ports which are minimal with respect to \rightsquigarrow^* to be executed - if two or more such ports exist, it follows that they are incomparable with respect to \rightsquigarrow^* and hence independent. Moreover, the hypothesis ensures that execution of such independent ports commute.

We will show that the execution within a whole step is confluent. By contradiction, and without loss of generality assume that there exist two distinct terminal states (m_1, u_1) , (m_2, u_2) reachable from the same initial state (m_0, u_0) . By terminal state we mean either a deadlock configuration or a state from which only the *sync* transition is possible.

Consider the graph of all possible executions from (m_0, v_0) within one step. Let us remark that this graph is finite and acyclic – since by construction we know that every port can be executed at most once within one step. On this graph, let us define the following subsets of states:

$$X_1 = \{(m, v) \mid (m, v) \rightarrow^* (m_1, v_1) \text{ and } \neg(m, v) \rightarrow^* (m_2, v_2)\}$$

$$X_2 = \{(m, v) \mid (m, v) \rightarrow^* (m_2, v_2) \text{ and } \neg(m, v) \rightarrow^* (m_1, v_1)\}$$

Intuitively, X_1 (resp. X_2) contains the states that lead eventually to the terminal state (m_1, v_1) (resp. (m_2, v_2)). Obviously, we have $X_1 \cap X_2 = \emptyset$. Moreover, we can prove that X_1 and X_2 are a partition of all the reachable states from (m_0, v_0) . By contradiction, assume there are states which are neither in X_1 or X_2 . Among them, we can choose one state (m, v) which has all successors in X_1 union X_2 – because we consider that there are precisely two terminal states. Now, if (m, v) has all successors in X_1 then it will eventually lead to (m_1, v_1) , so it belongs to X_1 – contradiction. The dual reasoning applies when (m, v) has all successors in X_2 . The only remaining possibility is that (m, v) has distinct successors into X_1 and X_2 respectively. So, let assume that $(m, v) \xrightarrow{p_1} (m'_1, v'_1)$ such that $(m'_1, v'_1) \in X_1$ and $(m, v) \xrightarrow{p_2} (m'_2, v'_2)$ such that $(m'_2, v'_2) \in X_2$. But, transitions p_1 and p_2 interleave – hence there exists (m'_{12}, v'_{12}) such that $(m'_1, v'_1) \xrightarrow{p_2} (m'_{12}, v'_{12})$ and $(m'_2, v'_2) \xrightarrow{p_1} (m'_{12}, v'_{12})$. This implies (m'_{12}, v'_{12}) belongs to both X_1 and X_2 , which is impossible because $X_1 \cap X_2 = \emptyset$.

Finally, since X_1 and X_2 define a partition of the reachable states from (m_0, v_0) in one step, we obtain the contradiction: the state (m_0, v_0) belongs to either X_1 or X_2 so, it cannot lead to both terminal states (m_1, v_1) and (m_2, v_2) .

□

4.4 The Synchronous BIP Language

We implement the Synchronous BIP components using the Synchronous BIP language which is an extension of the BIP language. The Synchronous BIP language (S-BIP) provides constructs for describing synchronous systems conforming to the formal framework as described up to now. The new constructs that S-BIP adds to the BIP language are described below:

- *modal type*: It specifies the behavior of an atomic component. For Synchronous BIP components, behavior is described by modal flow graphs (MFGs). MFGs are described by ports and dependencies between ports.
- *dependency type*: It specifies the dependency that relates ports and specify causal order between them. Dependencies can be of the following types: conditional, weak or strong.

A modal type is characterized by its ports and its dependencies. The syntax of a modal type is given below:

```

<modal type> ::= modal type <modal name>
                <variable definition>
                <port definition>
                <dependency definition>
            end

```

Variables are declared as data objects and they are typed, as in C language. They may have initial values. *Ports* are instances of port types, i.e. ports associated with typed variables. For a dependency, the definition is given below:

```

<dependency definition> ::= on < port name>
                           (<dependency type> (<port name>))*
                           (provided <expression>)? (do <expression>)?

```

A dependency is defined on a *port name* following the keyword **on**. The cause that triggers this port is given by the *port name* following the construct *<dependency type>*. The guard of the port is specified after the **provided** keyword and the action statement of the port is specified following the **do**.

The syntax for the dependency type is given below:

```

<dependency type> ::= <= | < - | < --

```

Dependency can be of type strong (\leq), weak ($\leq -$) or conditional ($\leq --$).

Example 25 The S-BIP description of the synchronous component *tick-tock* of Figure 4.10 is illustrated below:

```

modal type TickTock
  export port DataPort update(x)=update
  export port DataPort tock(x)=tock
  export port EventPort tick = tick

  on tick
  on tock <- tick provided x=60
    do x=0;
  on update <= tick <-- tock
    do x=x+1;
end

```

Ports *update*, *tock* and *tick* are exported at the interface of the component. The order of execution of the ports is defined by the dependencies and the associated guards. The execution of the component begins from *tick*, since its execution is not dependent on any other port.

The composition of two *tickTock* synchronous atomic components shown in Figure 4.12, is described in S-BIP as follows. The S-BIP description of this compound component is shown below:

```

compound type Clock
  component TickTock sec
  component TickTock min

  connector BroadcastData gtoctic(sec.tock1, min.tick2)
  connector SingletonEvent gs1(sec.tick1)
  connector SingletonEvent gs2(min.tock2)

  export port DataPort tick1 is gs1.tick1
  export port DataPort tock2 is gs2.tock2
end

component Clock clk

```

end

The compound component *Clock* is defined by two instances of component type *tickTock*, *sec* and *min* respectively. The components are composed via the connector *BroadcastData* which connects two ports of type *EventPort*. The singleton connectors *singleTick* and *singleTock* export the involved ports at the border of the compound component.

4.5 Related Work

The work we presented in this chapter is related to approaches with similar objectives. In the 42 framework [55] steps are described by using automata with final states. Another similarity is the distinction between data ports and control ports. Nonetheless, the latter are activated by controllers which are specific components. The synchronous/reactive domain of the Ptolemy system-level design framework [38] allows component-based description of synchronous systems where synchronous execution is orchestrated by a *director*. Finally, our work has the same general objectives as [17] which studies a compositional framework for heterogeneous reactive systems. In contrast to BIP, the framework is denotational and is based on the concept of tags marking the events of the signals of a system.

There are several differences between our work and existing results. Our work is based on operational semantics. It considers synchronous component-based systems as a particular case of the BIP framework which also encompasses general asynchronous computation. As we will show in the next chapter, our framework is expressive enough to allow modular translation of synchronous languages into BIP by preserving the structure of the source.

Modal flow graphs without data and only strong dependencies are acyclic partial orders on events. They correspond to acyclic marked graphs which are Petri nets without forward and backward conflicts.

Modal flow graphs with strong dependencies and their composition operation are also similar to *synchronous structures* used in a study of the synchronous model of computation [59]. This model has also some similarities with models such as *modal automata* [50] which distinguish between *must* and *may* transitions or *live sequence charts* [35] which distinguish between *hot* and *cold* events. Nonetheless, modal flow graphs encompass three independent modalities which are all necessary for modular description of synchronous systems. Furthermore, for a reasonably general class of modal flow graphs we proposed sufficient conditions for deadlock-freedom and confluence.

Conditional Dependency Graphs

Conditional Dependency Graphs (CDGs) [51, 39] is a tool that has been developed for compiling and implementing SIGNAL programs on given architectures. CDGs are labelled directed graphs that express clock inclusion and causality, where:

- *Vertices* are signals and clock variables
- *Edges* represent dependence relations. For x_1, x_2 two signals, the relation $x_1 \rightarrow x_2$ means that x_2 depends on x_1 .
- *Labels* represent the clocks at which the dependence relations are valid. For $x_1 \xrightarrow{h_c} x_2$ with h_c the clock of c , it means that x_2 depends on x_1 and the c is present.

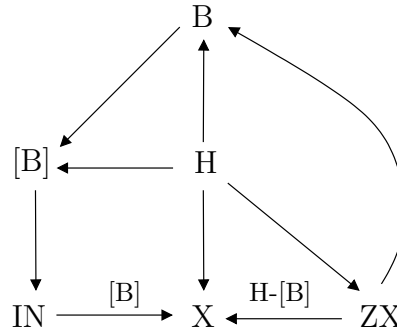


Figure 4.16: The conditional dependency graph for the example 14 of Chapter 3

Figure 4.16 shows the conditional dependency graph for the SIGNAL example 14 of Chapter 3. This graph can be interpreted as follows. The clock H is independent of any other clock. The possible execution of clocks is then $ZX, B, [B], IN$. The execution of X is restricted by the labels $[B]$ and $H - [B]$ on the edges (IN, X) and (ZX, X) respectively. That is, X depends on IN if $[B]$ is present, otherwise, if $H - [B]$ is present, on ZX .

As already mentioned, the BIP framework is used as a unifying semantic model for structural representation of different domain specific languages and programming models. The Synchronous BIP model which is a subset of BIP, was created in an attempt to integrate synchronous formalisms within BIP. Synchronous BIP is not a new synchronous formalism but a model for describing already existing synchronous formalisms. Currently, two synchronous formalisms have been represented by the Synchronous BIP model, LUSTRE and MATLAB/Simulink. On the other hand, SIGNAL is a synchronous programming language for real time systems development. Proof system, compilation and distributed implementation are some of the features it provides [22].

An attempt to compare Synchronous BIP and SIGNAL is described in the sequel. In modal flow graphs, vertices are ports or action names, attached with computation on variables (guards and update functions). Vertices in conditional dependency graphs are signals or typed variables and clock variables. Edges in modal flow graphs express dependencies between ports that define order of execution. There are three types of dependencies, strong, weak and conditional. In conditional dependency graphs, there exists only one type of edge which expresses dependence relation between a pair of signals. An edge can be labeled with a clock which restricts the dependency according to the presence or not of the clock. Finally, composition of modal flow graphs is defined through interactions between ports. In conditional dependency graphs composition is achieved with communication through common signals.

A representation of modal flow graphs in SIGNAL and conditional dependency graphs is shown in Figure 4.17. Ports p and q are attached with operations op_p and op_q and associated through causal dependencies. Each dependency is represented by a SIGNAL program and a conditional dependency graph. Both together, express the control and data dependency between the ports. For strong dependencies, ports p and q operate on the same clock, that is the clocks h_p and h_q must be equal. Moreover, the order of the operations on ports is op_p followed by op_q . For weak dependencies, the clock domain of port p is included in the time domain of q . Moreover, if the clock of q is present there is a data dependency between the operators of p and q such that, first is executed op_p and then op_q . Finally, for conditional dependencies, ports p and q are not clock related. The only constraint is set when both ports are executed on the same step. Then, because of data dependency, the operation on q proceeds the operation on p .

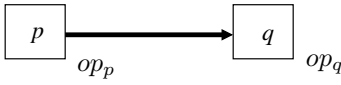

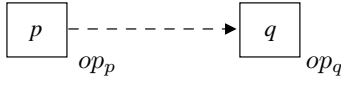
Dependency	Modal Flow Graphs	SIGNAL and Conditional Dependency Graphs	
		<i>Control flow</i>	<i>Data flow</i>
<i>strong</i>		$(h_p \wedge = h_q \mid op_p \mid op_q)$	$op_p \longrightarrow op_q$
<i>weak</i>		$(h_q \wedge < h_p \mid op_p \mid op_q)$	$op_p \xrightarrow{h_p} op_q$
<i>conditional</i>		$(op_p \mid op_q)$	$op_p \xrightarrow{h_p \wedge * h_q} op_q$

Figure 4.17: Relation of Modal Flow Graphs with SIGNAL/Conditional Dependency Graphs

From the above description, modal flow graphs can clearly be considered as a subset of the conditional dependency graphs. All three dependencies can be successfully expressed using the SIGNAL syntax and the conditional dependency graphs. Moreover, the syntactic restriction of modal flow graphs, known as well-triggered, can be captured by the notion of endochrony proposed by SIGNAL. Endochronous SINGAL models have a unique way to compute the clocks of signals, preserving in that way deterministic behavior. However, dependencies in modal flow graphs are expressed on occurrence of ports and not on their absence. On the other hand, conditional dependency graphs treat in the same way absence and presence of clocks. Finally, modal flow graphs have one to one mapping to Petri nets which is not yet clearly presented for SIGNAL programs.

4.6 Conclusion

In this chapter we presented Synchronous BIP as an extension of the BIP component-based framework. Synchronous BIP is a formalism for modeling synchronous data flow systems. Behavior of Synchronous BIP components is described by modal flow graphs. These are a particular class of Petri nets for which deadlock-freedom and confluence are met by construction provided some easy-to-check conditions hold.

In the following two chapters, we provide translations of synchronous formalisms into Synchronous BIP. The first concerns the translation of the Lustre language and the second the translation of MATLAB/Simulink. These translations show the interplay between data flow and control flow and allow understanding how strict synchrony can be weakened to get less synchronous computation models.

Chapter 5

Language Factory for Synchronous BIP

This chapter provides two transformations from synchronous formalisms into Synchronous BIP. The first concerns the transformation of the LUSTRE language and the second the transformation of the discrete-time fragment of MATLAB/Simulink. For both methods we present the principles of the translations and we give the modal flow graphs that correspond to the LUSTRE operators and to several Simulink blocks respectively. Both translations are fully implemented for automatic generation of Synchronous BIP code from LUSTRE and Simulink models respectively. For LUSTRE we provide theoretical results on the correction of the translation. For MATLAB/Simulink we validate the translation to Synchronous BIP based on experimental results. We conclude this chapter with a discussion concerning the translations of more synchronous formalisms into Synchronous BIP.

This chapter is structured as follows. Section 5.1 describes the translation from LUSTRE to Synchronous BIP. The translation from MATLAB/Simulink to Synchronous BIP is described in section 5.2. Section 5.3 draws some conclusions.

5.1 From LUSTRE to Synchronous BIP

For the translation of Lustre programs into Synchronous BIP, we consider statically correct programs which satisfy the static semantics rules of Lustre [41]. These rules exclude programs containing cyclic, dependent equations, recursive calls of nodes as well as combinatorial operators applied to expressions having different clocks. We define modular translation for LUSTRE to Synchronous BIP, first for single-clock programs and then for multi-clock programs.

5.1.1 Principles of the Translation for Single-clock LUSTRE Nodes

The single-clock subset of Lustre is generated by using only *combinatorial* and *unit delay* operators. All flows are sampled (indexed) by the basic clock.

The translation from the single-clock subset of Lustre to Synchronous BIP is modular. Each Lustre node is represented by a *well-triggered* Synchronous BIP component with two kinds of ports:

1. *control port*, including a unique *act* port which is triggered by the basic clock and initiates the step of the node.

2. *data ports*, including $in_1, \dots, in_i, out_1, \dots, out_j$ for *input* events and *output* events respectively. These data ports carry data input (resp. output) that are read (resp. written) by the node.

Additionally, atomic Synchronous BIP components may contain internal ports and variables, depending on the specific computation carried by the node.

The interface of a Synchronous BIP component associated to a Lustre node is shown in Figure 5.1

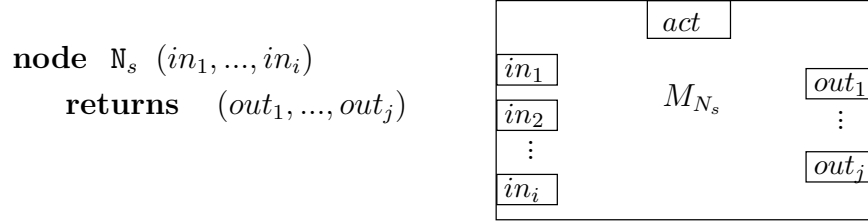


Figure 5.1: A single-clock LUSTRE node N_s and its associated Synchronous BIP component M_{N_s}

The Synchronous BIP component representing a single-clock Lustre node is obtained by corresponding to each of its content elements an atomic Synchronous BIP *component* and by composing them using a set of *interactions*. These two steps are defined below:

- *components*: For each single-clock operator, we add a Synchronous BIP atomic component. Moreover for each call of subnode within the equations, we add its corresponding Synchronous BIP component.
- *interactions*: Interactions are of two types:
 1. *control flow* interaction: it realizes strong synchronization between all the act ports of all components. A Synchronous BIP component representing a single-clock Lustre node has only one control flow interaction.
 2. *data flow* interaction: it synchronizes one *out* port to one or more *in* ports. They are used to propagate data from input flow components to expression components, between different expression components and from expression components to output flow components, according to the syntactic structure of expressions and equations.

5.1.2 Translation of Single-clock LUSTRE Operators

The Synchronous BIP components shown in figure 5.2 correspond to a data *flow*, a *pre* operator and a *combinatorial* operator respectively.

The *flow* component whenever activated through the *act* port, reads a value x through the *in* port and outputs this value through the *out* port in the same execution step. The *pre* component has a local variable which is initially set to x_0 . Whenever it is activated through *act*, it outputs the current value x , then it reads and assigns a new value to x to be used in the next step. The *combinatorial* component starts a step when it is triggered through the *act* port. Then it reads input values in some arbitrary order, performs its specific computation, and finally produces an output value. The corresponding synchronous BIP code is shown in Figure 5.3.

Example 26 An integrator node in LUSTRE (left) and its corresponding network of operators (right) are shown in Figure 5.4.

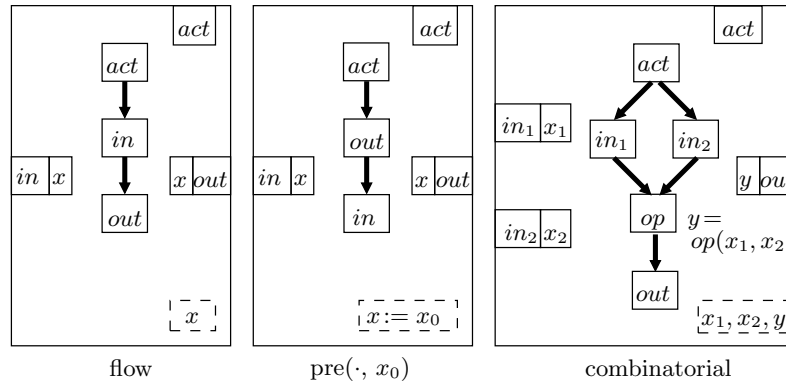


Figure 5.2: Single-clock operators

modal type Pre

```

data int x=0
export port EventPort act=act
export port DataPort in(x)=in
export port DataPort out(x)=out

on act
on out <= act
on in <= out
end

```

modal type Plus

```

export port EventPort act=act
export port DataPort in1(x1)=in1
export port DataPort in2(x2)=in2
export port DataPort out(y)=out
port EventPort op

on act
on in1 <= act
on in2 <= act
on op <= in1 in2
do y=x1+x2;
on out <= op
end

```

Figure 5.3: The synchronous BIP code for a **pre** operator (left) initialized to zero and a **plus** operator (right) for adding two integers

```

node Integrator(i: int)
returns o: int;
let o = i + pre(o,0); tel;

```

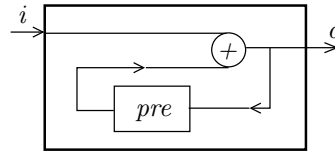


Figure 5.4: An integrator described in LUSTRE

Figure 5.5 illustrates the produced Synchronous BIP for the integrator node shown above, as a composition of atomic components corresponding to elementary operators in LUSTRE.

The atomic components correspond to the pre expression, the combinatorial expression for the + operator, the input flow i and the output flow o . There is a unique control flow interaction γ_{act} that strongly synchronizes the act ports of all components. There are also data flow interactions for data transfer from outputs to inputs and which are the following: 1) g_1 , from the input flow component to the + component, 2) g_2 , from the pre component to the + component and 3)

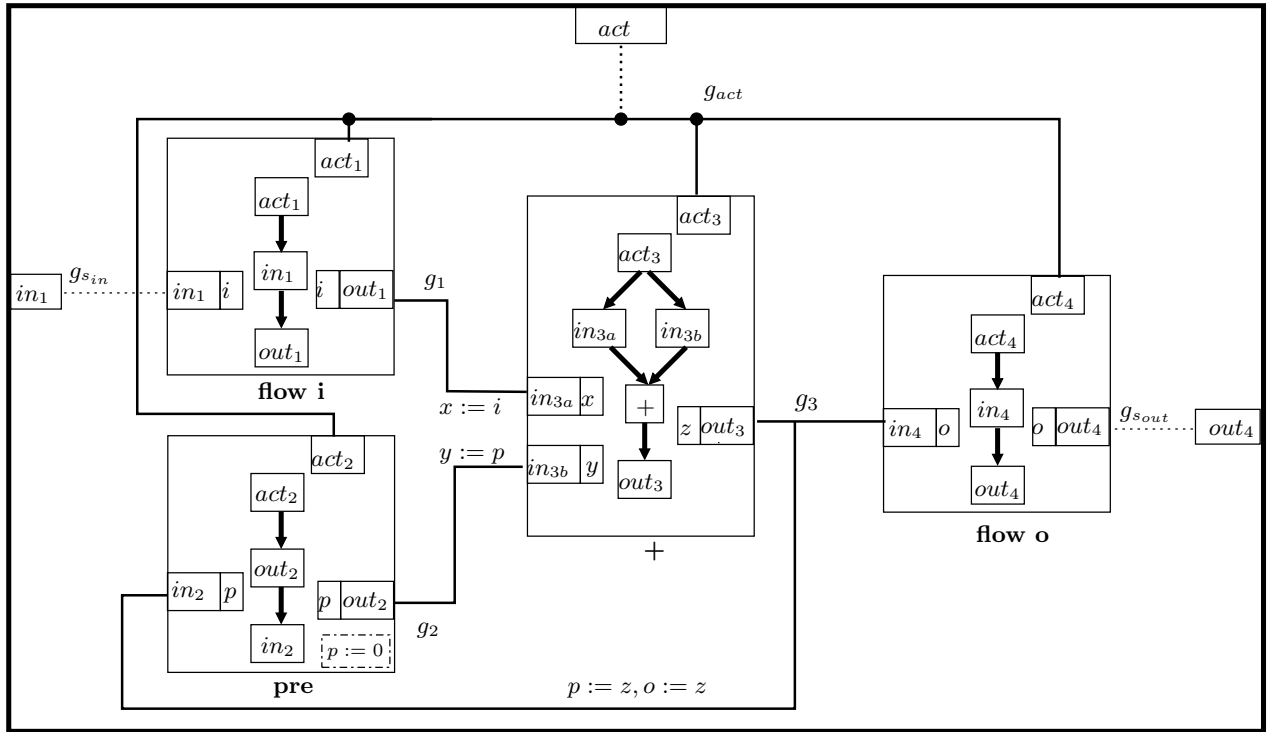


Figure 5.5: The *integrator* node of Figure 5.5 described as a compound synchronous BIP component

g_3 , from the $+$ component to the output flow component and back to the pre component. The corresponding synchronous BIP code is shown in Figure 5.6

The compound component corresponding to the integrator, is shown in Figure 5.7. All control ports are composed in one port called $act_1act_2act_3act_4$ and which represents all ports participating on the g_{act} connector. Ports involved in the interactions g_1, g_2, g_3 are mapped to a port each, $out_1in_{3a}, out_2in_{3b}$ and out_3in_2 – 4 respectively. Ports in_1 and out_4 are the points of communication of the component with its environment.

Example 27 Figure 5.8 (left) shows the LUSTRE node for a watchdog device and its corresponding synchronous network of operators (right).

Figure 5.9 shows its representation in Synchronous BIP as composition of atomic components. Each operator is mapped to an atomic synchronous BIP component. The link between the operators are mapped to connectors $g_1...g_7$. All components are strongly synchronized through the g_{act} connector. The composite component communicates with its environment through the ports exported by the connectors $g_{s1}, g_{s2}, g_{s3}, g_{s4}$ and g_{act} . The Synchronous BIP code for the watchdog device is illustrated in Figure 27.

5.1.3 Principles of the Translation for Multi-clock LUSTRE Nodes

The multi-clock subset of Lustre is generated using two additional operators, the *sampling* operator and the *interpolation* operator.

The translation from the multi-clock subset of Lustre to Synchronous BIP is modular. Each multi-clock Lustre node is represented by a *well-triggered* Synchronous BIP component with two kinds of ports:

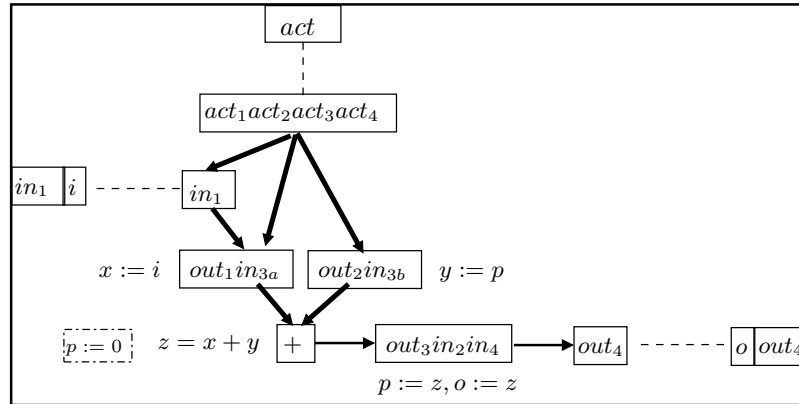
```

modal type IntegratorCompound
  component Flow = flow_i
  component Flow = flow_o
  component Pre = pre
  component Plus =plus

  connector RendezVous4Events  $g_{act}$ (flow_i.act1,
    flow_o.act4, pre.act2, plus.act3)
  connector RendezVousData  $g_1$ (plus.in3a, flow_i.out1)
  connector RendezVousData  $g_2$ (plus.in3b, pre.out2)
  connector RendezVous3Data  $g_3$ (flow_o.in4,
    pre.in2,plus.out3)
  connector SingletonData  $g_{sin}$ (flow_i.in1)
  connector SingletonData  $g_{sout}$ (flow_o.out4)

  export port DataPort in is  $g_{sin}.in_1$ 
  export port DataPort out is  $g_{sout}.out_4$ 
  export port EventPort act is  $g_{act}.act$ 
end

```

Figure 5.6: The synchronous BIP code for the *integrator* of Figure 5.4Figure 5.7: The *integrator* synchronous BIP composite component produced as the composition of atomic Synchronous BIP components

1. *control ports*, including act , act_a , ..., act_z ports which are triggered by the basic clock and slower (derived) clocks. Slower (derived) clocks correspond to the LUSTRE expression **when** b , for $b = a, \dots, z$, where b is a boolean variable.
2. *data ports*, including $in_1, \dots, in_i, out_1, \dots, out_j$ for *input* events and *output* events respectively. These data ports carry data input (resp. output) that are read (resp. written) by the node.

Additionally, atomic Synchronous BIP components may contain internal ports and variables, depending on the specific computation carried by the node.

The interface of a Synchronous BIP component associated to a multi-clock Lustre node is


```

node Watchdog(set, reset, deadline: bool)
  returns alarm: bool;
var is_set: bool;
let
  alarm = deadline and is_set;
  is_set = set -> if set then true
                  else if reset then false
                  else pre(is_set);
tel.

```

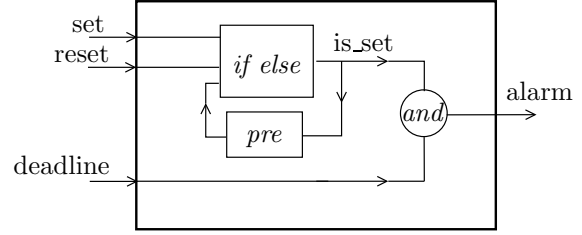


Figure 5.8: A *watchdog* in LUSTRE (left) and as a synchronous network of operators (right)

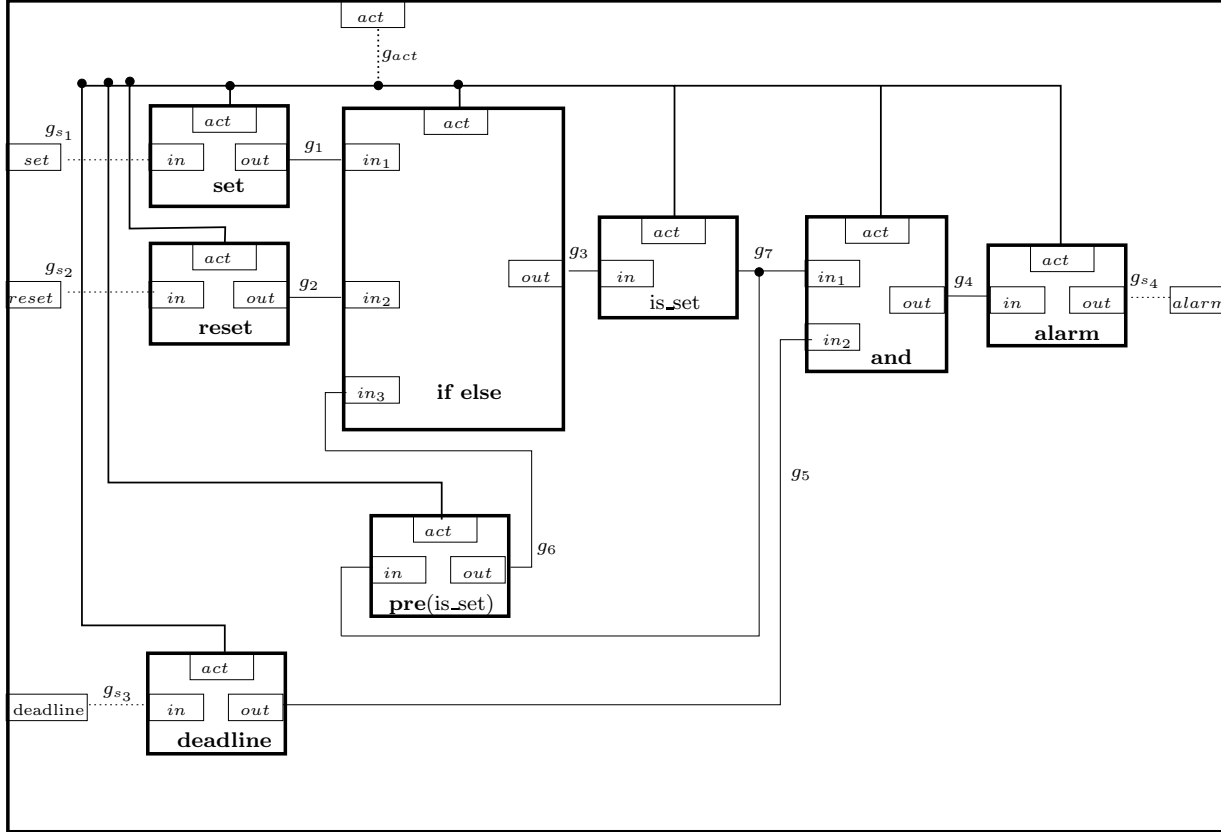


Figure 5.9: A *watchdog* device described as the composition of Synchronous BIP components

shown in Figure 5.11

The method we apply for building Synchronous BIP components for multi-clock nodes is similar to the one used for single-clock nodes. It consists of two steps, first, corresponding basic Lustre elements to atomic Synchronous BIP components and second, composing atomic components by using a set of interactions. These steps are described in more details as follow.

- *components*: First, we add a *derived clock* component for each clock. The derived clock component corresponds to the Lustre expression **when** b . Second, we add a *sampling* (resp. *interpolation* component for each sampling (resp. interpolation) expression occurring within the equations of a Lustre node. All other elements and equations occurring in a Lustre node are translating as in the single-clock case.

```

modal type Watchdog
  component Flow = set
  component Flow = reset
  component Flow = deadline
  component Flow = is_set
  component Flow = alarm
  component Pre = pre
  component And = and
  component Conditional =if_else

  connector RendezVous8Events  $g_{act}$ (set.act,
    reset.act, deadline.act, is_set.act
    alarm.act, and.act, pre.act, if_else.act)

  connector RendezVousData  $g_1$ (if_else.in1, set.out)
  connector RendezVousData  $g_2$ (if_else.in2, reset.out)
  connector RendezVousData  $g_3$ (is_set.in, if_else.out)
  connector RendezVousData  $g_4$ (alarm.in, and.out)
  connector RendezVousData  $g_5$ (and.in2, deadline.out)
  connector RendezVousData  $g_6$ (if_else.in3, pre.out)
  connector RendezVous3Data  $g_7$ (and.in1,
    pre.in, is_set.out)

  connector SingletonData  $g_{s_1}$ (set.in)
  connector SingletonData  $g_{s_2}$ (reset.in)
  connector SingletonData  $g_{s_3}$ (deadline.in)
  connector SingletonData  $g_{s_4}$ (alarm.out)

  export port DataPort set is  $g_{s_1}.in$ 
  export port DataPort reset is  $g_{s_2}.in$ 
  export port DataPort deadline is  $g_{s_3}.in$ 
  export port DataPort alarm is  $g_{s_4}.out$ 
  export port EventPort act is  $g_{act}.act$ 
end

```

Figure 5.10: Synchronous BIP code for the *watchdog* of Figure 5.9

- *interactions*: The data flow interactions are the same as for the single-clock case, with the addition that data is also propagated to the input port of the derived clock. Regarding the control flow interactions, we add one interaction which synchronizes all the *act* ports of flows and expression sampled on the basic clock. Moreover, for each derived clock component, we add an interaction which synchronizes its *clock* port with all *act* ports of flows and expressions sampled under that slower clock.

The following theorem establishes the correctness of the translation from single-clock Lustre to Synchronous BIP. We consider statically correct programs which satisfy the static semantics rules of Lustre [41]. These rules exclude programs containing cyclic, dependent equations, recursive calls of nodes as well as combinatorial operators applied to expressions having different

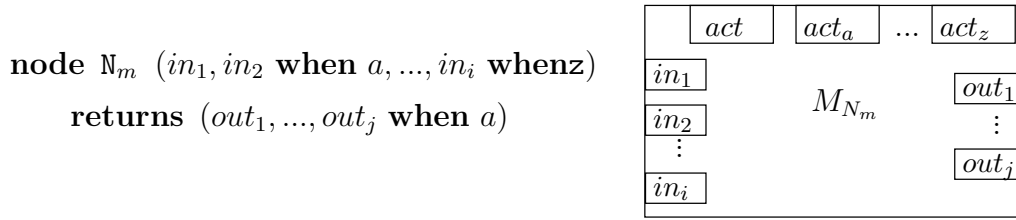


Figure 5.11: A multi-clock LUSTRE node N_m and its associated Synchronous BIP component M_{N_m}

clocks. However, applying the translation to statically incorrect Lustre programs, we obtain Synchronous BIP components which do not satisfy desirable properties of acyclic behavior and well-triggeredness.

The following theorem is a consequence of modularity of translation and of the following facts:

- The modal flow graphs corresponding to the basic constructs of Lustre are well-triggered;
- For statically correct Lustre programs [41], composition of the basic modal flow graphs preserves well-triggeredness.

Theorem 3 *Every statically correct single-clock Lustre node N is represented by a well-triggered S synchronous BIP component B_N^f such that:*

1. *it has a unique root which is an act port;*
2. *all its dependencies are strong;*
3. *it is deadlock-free and confluent;*
4. *simulates the micro-step Lustre semantics [41] of N .*

Proof.

1. Each atomic single-clock component has a unique root, the *act* port. By composition all *act* ports are strongly synchronized that leads to a component with a unique root.
2. From definition 17, dependencies are inherited by composition. Since all dependencies in atomic components are strong the conclusion follows.
3. *Deadlock-freedom.* Following theorem 1, B_N^f is deadlock-free, if each port with strong dependencies has its guard true. For each interaction, guards are obtained as conjunction of guards of sub-components. Given that in atomic components all guards are true, then the compound component has all its guards true and therefore is deadlock-free.

Confluence. Following theorem 2, a Synchronous BIP component is confluent, if the result of the execution of a step is independent of the order of the execution of independent ports. The synchronous BIP component B_N^f is confluent, if for every independent ports $p_1 \# p_2$, their associated guarded actions are independent. For the independent ports p_1, p_2 one of the following two situations may happen: 1) p_1 (resp. p_2) is port of the atomic component B_1^f (resp. B_2^f) or 2) both p_1 and p_2 are ports of the same atomic component B^f . In both cases the actions associated to the ports are independent. For the first case, all

actions are defined on disjoint sets of variables. For the second case, p_1 and p_2 can only be in data ports of a combinatorial component and have associated different variables by construction.

4. The possible executions of B_f^N are determined by dependencies and interactions between atomic components. Initially, all components are triggered by the control act interaction and they all complete a step till the next activation. The order of execution of data ports is constrained by strong dependencies within atomic components and by data interactions. The former enforces local constraints e.g., a) *flow* components update their values, then deliver them, b) *pre* components deliver their values, then they are updated, c) *combinatorial* operator components update all their inputs, then compute and deliver the result. Data flow interactions enforce overall data-flow constraints e.g., a) the results of sub-expressions are required to evaluate expressions, b) the right-hand expressions in equations are required to update output flows, etc. These constraints restrict as little as possible the order of actions while ensuring the correct operation within a synchronous step.

□

5.1.4 Translation of Multi-clock LUSTRE Operators

In Figure 5.12 we provide two atomic components, the *sampling* component and the *interpolation* component which model the sampling and interpolation operation of Lustre respectively and the *derived clock* component which generates slower clocks corresponding to a boolean flow b . The corresponding synchronous BIP code for the last two atomic components is shown in Figure 5.1.4.

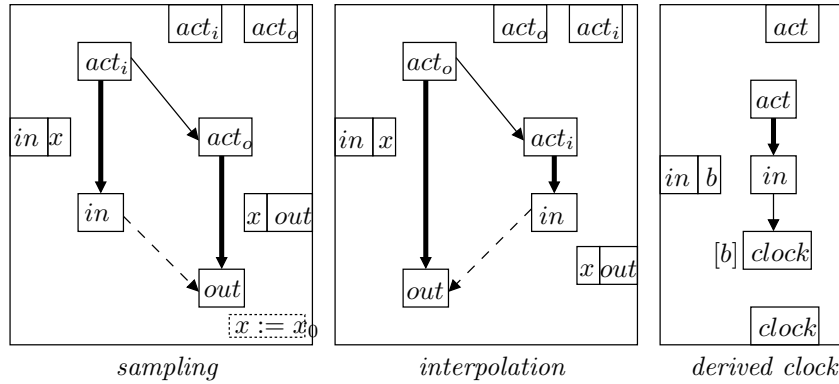


Figure 5.12: Atomic components for multi-clock Lustre operators

Both of the first two components have two control ports act_i and act_o triggering respectively the input in and the output out data flow ports. For a *sampling* component, act_o depends weakly on the act_i , and moreover, the output port out depends conditionally on the input port in . Thus an input is always read and whenever required, an output is produced with the most recent value of the input. For the *interpolation* component, we have the opposite: act_i depends weakly on act_o but out depends conditionally on in . Thus the input is read on specific instants but the output is always produced with the most recent value of the input. The last atomic component is used to initiate all the computations carried on the clock b . Intuitively, it triggers the $clock$ port only after its base clock act has been triggered and if the value b read through the data flow port in is true.

```

modal type Interpolation
  export port EventPort act_o=act_o
  export port EventPort act_i=act_i
  export port DataPort in(x)=in
  export port DataPort out(x)=out

  on act_o
  on act_i <-act_o
  on in <= act_i
  on out <= act_o <-- in
end

modal type Derived_clock
  export port EventPort act=act
  export port EventPort clock=clock
  export port DataPort in(b)=in

  on act
  on in <= act
  on clock <-in provided b
end

```

Figure 5.13: The synchronous BIP code for an **interpolation** operator (left) and a **derived clock** (right)

```

node input_handler(a: bool, x: int when a)
returns y: int;
let y = if a then current x else pre(y, 0);
tel ;

node output_handler(c: bool, y: int)
returns z: int when c;
var yc: int when c;
let yc = y when c; z = yc * yc ;
tel ;

node input_output(a,c: bool, x: int when a)
returns z: int when c;
var y: int;
let y = input_handler(a, x);
    z = output_handler(c, y);
tel;

```

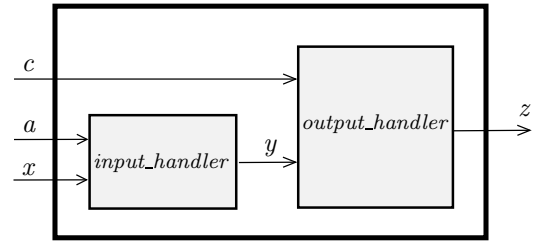


Figure 5.14: Input/output handler in LUSTRE

Example 28 The input/output handler of Figure 5.14 is a multi-clock LUSTRE node.

The main node is the input/output and uses two other nodes, the `input_handler` and the `output_handler`. Figure 5.15 shows the synchronous BIP composite component for the `input_handler` node. It is the composition of atomic components that correspond to single-clock operators (flow a , flow x ,...) and to multi-clock operators (current x ,...).

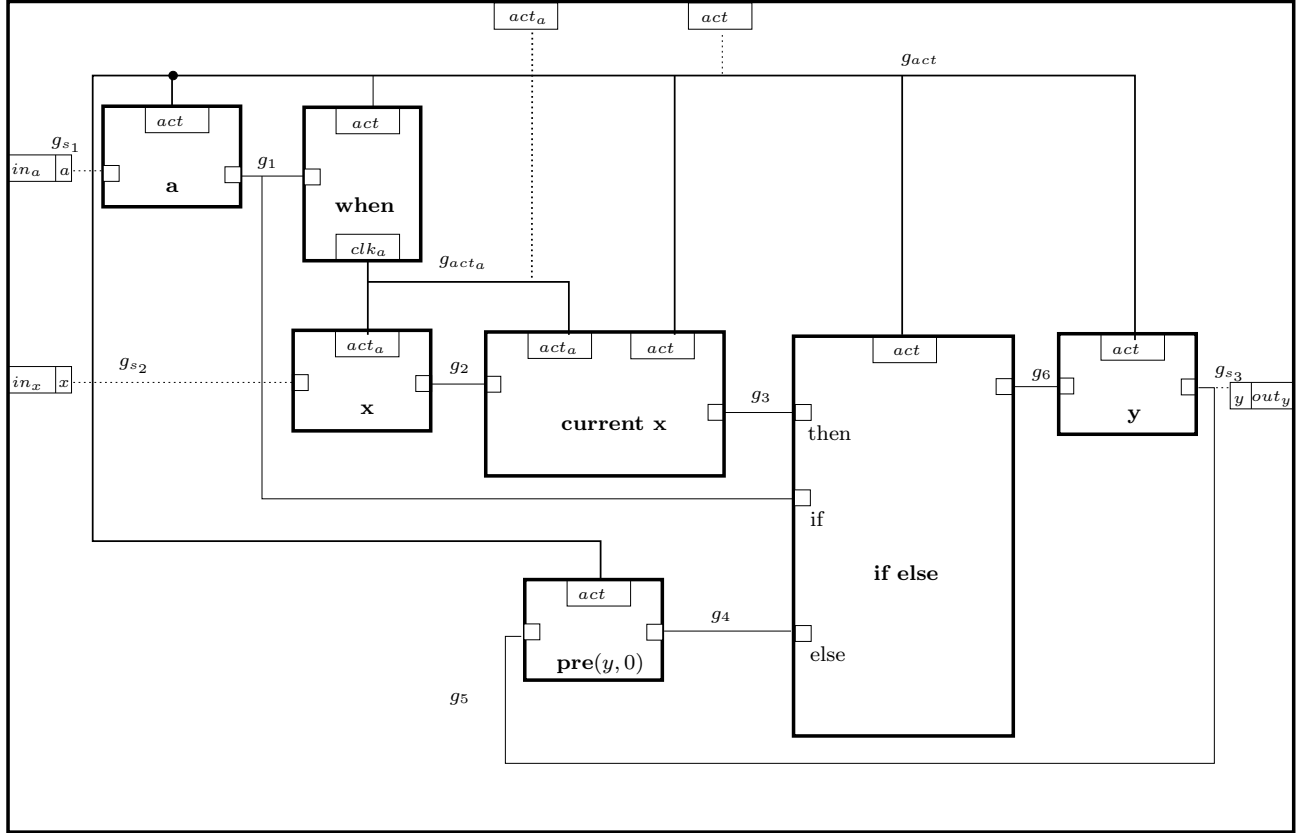


Figure 5.15: The synchronous BIP composite component for the LUSTRE `input_handler` node

The composite component exports at its interface two control ports, act that correspond to the basic clock and act_a that represents a slower clock a . The clock clk_a is produced by the component **when** if the value of a is true. The g_1, \dots, g_6 are strong synchronizations between ports for data transfer between different atomic components. Finally, the component exports also data ports at its interface, in_a, in_x and out_y to each of which is assigned a data variable a, x and y respectively.

Example 29 Figure 5.16 shows the synchronous compound component for the input/output handler. The `input_handler` component receives the data a and x at the ports in_a and in_x respectively. It communicates with the `output_handler` through the port out_y propagating the variable y . The `output_handler` receives also the variable c and produces z at the port out_z . The activation ports act, act_a , and act_c correspond respectively to the basic, **when** a and **when** c clocks. The synchronous BIP code for the composite compound component input/output handler is shown below:

modal type InOutHandler

component input_handler = in_handler

```

component output_handler = out_handler
connector RendezVous2Events gact(in_handler.act,
    out_handler.act)
connector RendezVousData gio(out_handler.iny, in_handler.outy)
connector SingletonData gs1(in_handler.inc)
connector SingletonData gs2(in_handler.ina)
connector SingletonData gs3(in_handler.inx)
connector SingletonData gs4(out_handler.outz)
connector SingletonEvent gacta(in_handler.acta)
connector SingletonEvent gactc(out_handler.actc)

export port DataPort inc is gs1.inc
export port DataPort ina is gs2.ina
export port DataPort inx is gs3.inx
export port DataPort outz is gs4.outz
export port EventPort act is gact.act
export port EventPort acta is gacta.acta
export port EventPort actc is gactc.actc

end

```

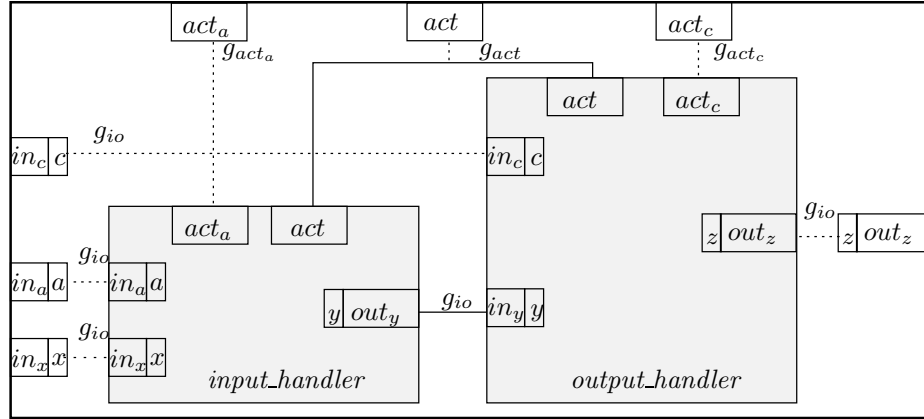


Figure 5.16: Compound component for the input/output handler of Figure 5.14

The synchronous BIP component produced by the composition of the components *input_handler* and *output_handler* is shown in Figure 5.17. It is decomposed into three subgraphs each of which is rooted by one of the activation ports *act*, *act_a*, and *act_c*.

The following theorem that establishes the correctness of the translation from multi-clock Lustre to Synchronous BIP. We consider statically correct programs which satisfy the static semantics rules of Lustre [41].

Theorem 4 *Every statically correct single-clock Lustre node N is represented by a well-triggered S synchronous BIP component B_N^f such that:*

1. *it has a unique root which is an act port;*

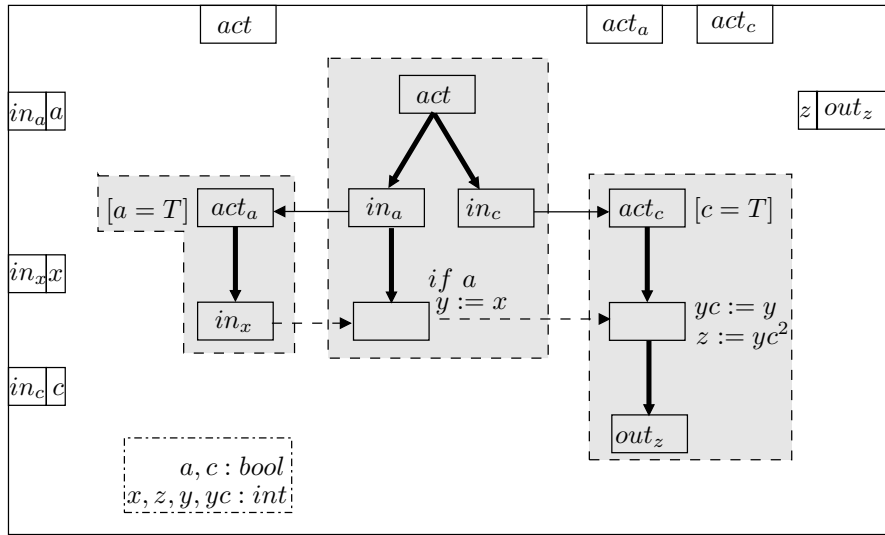


Figure 5.17: The Synchronous BIP component corresponding to the input/output handler of Figure 5

2. all its dependencies are strong;
3. it is deadlock-free and confluent;
4. simulates the micro-step Lustre semantics [41] of N .

Proof.

1. Each atomic single-clock component has a unique root, the act port. By composition all act ports are strongly synchronized that leads to a component with a unique root.
2. From definition 17, dependencies are inherited by composition. Since all dependencies in atomic components are strong the conclusion follows.
3. *Deadlock-freedom.* Following theorem 1, B_N^f is deadlock-free, if each port with strong dependencies has its guard true. For each interaction, guards are obtained as conjunction of guards of sub-components. Given that in atomic components all guards are true, then the compound component has all its guards true and therefore is deadlock-free.

Confluence. Following theorem 2, a Synchronous BIP component is confluent, if the result of the execution of a step is independent of the order of the execution of independent ports. The synchronous BIP component B_N^f is confluent, if for every independent ports $p_1 \# p_2$, their associated guarded actions are independent. For the independent ports p_1, p_2 one of the following two situations may happen: 1) p_1 (resp. p_2) is port of the atomic component B_1^f (resp. B_2^f) or 2) both p_1 and p_2 are ports of the same atomic component B^f . In both cases the actions associated to the ports are independent. For the first case, all actions are defined on disjoint sets of variables. For the second case, p_1 and p_2 can only be in data ports of a combinatorial component and have associated different variables by construction.

4. The possible executions of B_f^N are determined by dependencies and interactions between atomic components. Initially, all components are triggered by the control act interaction

and they all complete a step till the next activation. The order of execution of data ports is constrained by strong dependencies within atomic components and by data interactions. The former enforces local constraints e.g., a) *flow* components update their values, then deliver them, b) *pre* components deliver their values, then they are updated, c) *combinatorial* operator components update all their inputs, then compute and deliver the result. Data flow interactions enforce overall data-flow constraints e.g., a) the results of sub-expressions are required to evaluate expressions, b) the right-hand expressions in equations are required to update output flows, etc. These constraints restrict as little as possible the order of actions while ensuring the correct operation within a synchronous step.

□

Theorem 5 *Every statically correct multi-clock Lustre Node N is represented by a well-triggered Synchronous BIP component B_N^f which:*

1. *has multiple (control) root act ports, one for each clock in the Lustre program, and multiple data in/out ports;*
2. *the subgraphs are defined by strong dependencies and are interconnected through weak dependencies forming a tree;*
3. *is deadlock-free and confluent;*
4. *simulates the micro-step Lustre semantics [41] of N .*

Proof.

1. Each atomic multi-clock component, except the derived clock component, has two roots, the act_i port and the act_o ports. The *clock* port of each derived clock component is synchronized with all act ports of components which are sampled by that clock. By composition, that leads to a component with multiple *act* ports, one for each clock. Each of these roots define a subgraph where all ports are sampled by the same clock. In addition, the component has multiple data *in/out* ports which derive from the multiple subgraphs and each of them may be sampled by a different clock.
2. At each atomic single-clock or multi-clock component, data ports are strongly dependent on *act* ports. The composition by synchronization of *act* ports, leads to components with subgraphs rooted by *act* interactions. By definition 17, chapter 4, the subgraphs inherit the strong dependencies which are defined between the control ports and the data ports. In addition, the *clock* port of the derived clock depends weakly on data ports triggered on faster clocks. Consequently, weak dependencies interconnect the subgraphs, by triggering roots sampled on slower clocks. That leads to an overall acyclic structure, that is, a set of trees. But, since the *act* interaction sampled on the basic clock does not have any dependencies, the structure reduces to a unique tree.
3. Similar to theorem 4, point 3.
4. Similar to theorem 4, point 4.

□

5.1.5 Implementation of the Translation

The translation from LUSTRE to Synchronous BIP has been implemented in the **Lustre2S-BIP** tool. It parses LUSTRE files (`.lus`) and produces Synchronous BIP models (`.bip`). The generated models reuse a (handwritten) predefined component library of atomic components and connectors (`lustre.bip`). This library contains the Lustre operators (combinatorial, **pre**, **when**, **current**,...) as well as the most useful connectors for data transfer and control activation. Chapter 6 presents LUSTRE node that were translated in Synchronous BIP.

5.2 From MATLAB/Simulink into Synchronous BIP

The modeling of Simulink models in Synchronous BIP is far from being trivial. The underlying models of computation are essentially different i.e., synchronous, step-based for Simulink and asynchronous, interaction-based for Synchronous BIP. Simulink uses very particular control execution mechanisms such as the triggering and enabling of sub-systems. It has informal semantics defined operationally through a simulation engine. The user can use simulation parameters (e.g. simulation step, solver used, etc) the meaning of which is only partially documented.

The translation from Simulink to synchronous BIP is modular and enjoys the same properties as the translation of Lustre. That is, each “correct” Simulink model is represented by a well-triggered component of Synchronous BIP which is always deadlock-free and confluent. The proposed translation exhibits maximal parallelism, that is, it enforces only the absolutely necessary dependencies between events needed for correct execution.

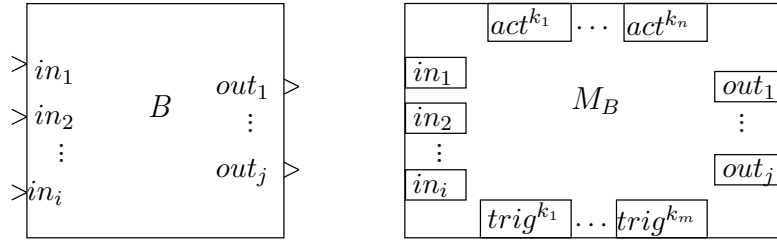
In the following paragraphs we present the principles of the translation as well as details for the translation of basic Simulink blocks and Simulink subsystems. The translation is restricted to Simulink blocks which are simulated using “fixed-step solver in single tasking mode”.

5.2.1 Principles of the Translation

The translation from Simulink models into Synchronous BIP associates with each Simulink model B a unique synchronous BIP component M_B . Moreover, basic Simulink blocks e.g., operators, are translated into elementary (atomic) synchronous BIP components. Structured Simulink blocks e.g., subsystems, are translated recursively as composition of the components associated to their contained blocks. The composition is also translated structurally i.e., dataflow and activation links used within the subsystem are translated to connectors.

To avoid confusion between control (resp.data) ports of Simulink with control (resp.data) ports of BIP we will refer to the latter as *control* (resp. *data*) *BIP ports*. The interface of Synchronous BIP components associated to Simulink blocks is shown in Figure 5.18. Such components involve two categories of BIP ports, control BIP ports and data BIP ports.

- *control BIP ports*, including $act^{k_1}, \dots, act^{k_n}$ and $trig^{k_1}, \dots, trig^{k_m}$ for *activation* ports and *triggering* events respectively. These BIP ports represent pure input and output control events. They are used to coordinate the overall execution of modal flow graph behavior and correspond to control mechanisms provided by Simulink e.g., sample times, triggering signals, enabling conditions, etc.
- *data BIP ports*, including in_1, \dots, in_i and out_1, \dots, out_j for *input* ports and *output* ports respectively. These BIP ports transport data values into and from the component. They are used to build the dataflow links provided by Simulink.

Figure 5.18: A Simulink block B and its associated Synchronous BIP component M_B

The synchronous BIP component M_B that corresponds to the Simulink model B needs an additional synchronous component Clk_B which generates all activation events $act^{k_1}, act^{k_2}, \dots$ that correspond to periodic sample times k_1, k_2, \dots used within the model. The final result of the translation will be the composition of M_B and Clk_B with synchronization on activation events.

5.2.2 Translation of Simulink Ports and Simulink Atomic Blocks

Simulink ports and atomic blocks are translated into elementary Synchronous BIP components. Each component has a single activation control BIP port act^k and several data BIP ports in_1, \dots, in_n, out . Computation of functions on the inputs is done on internal BIP ports. The control BIP port coordinates the execution of the graph and corresponds to the sample time k of the Simulink port/block. At each activation of act^k actual data values x_1, \dots, x_n are received on all input events in_1, \dots, in_n and the output value y is computed and sent on the output event out .

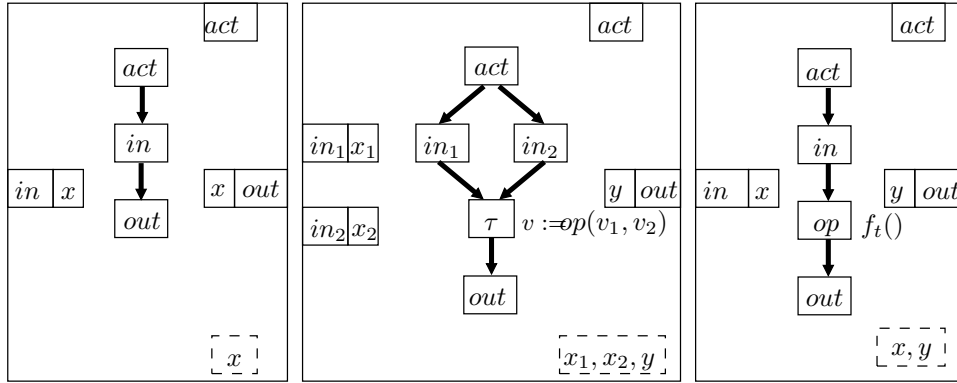


Figure 5.19: Atomic synchronous BIP components for Simulink data ports (left), combinatorial blocks (middle) and transfer functions (right)

Simulink *inports* and *outports* are translated into elementary synchronous BIP components as shown in figure 5.19 (left). These graphs represent a simple identity flow i.e., at each activation of the control BIP port act one value x of data comes in and goes out through the data BIP ports in and out respectively.

Combinatorial blocks are translated as shown in figure 5.19 (middle). At each activation of the control BIP port act , actual data values x_1 and x_2 are received on all input data BIP ports in_1 and in_2 and then the output value y is computed and then sent on the output data BIP port out .

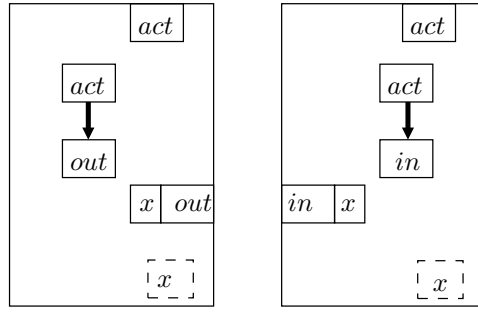


Figure 5.20: Atomic synchronous BIP components for Simulink source (left) and sink (right) blocks

Transfer functions are translated as shown in figure 5.19 (right). For a given transfer function $\frac{b_0z^0+\dots+b_qz^{-q}}{1+a_1z^{-1}+\dots+a_pz^{-p}}$ the computation is realized by the function $f_t()$ as follows:

$$\begin{aligned} r[0] &:= u \\ s[0] &:= \sum_{j=0}^q b_j r[j] - \sum_{i=1}^p a_i s[i] \\ r[j] &:= r[j-1] \text{ for all } j = q \text{ down to } 1 \\ s[i] &:= s[i-1] \text{ for all } i = p \text{ down to } 1 \\ v &:= s[0] \end{aligned}$$

where s and r are buffers for the input/output values.

Simulink *sources* and *sink* blocks are translated into elementary modal flow graphs as shown in figure 5.20. At each activation of the control BIP port *act*, these graphs produce (respectively consume) one data value y through the output data BIP ports *out* (respectively input data BIP port) *in*.

Figure 5.21 shows the synchronous BIP components corresponding to *unit-delay* and *zero-order hold* blocks of Simulink respectively.

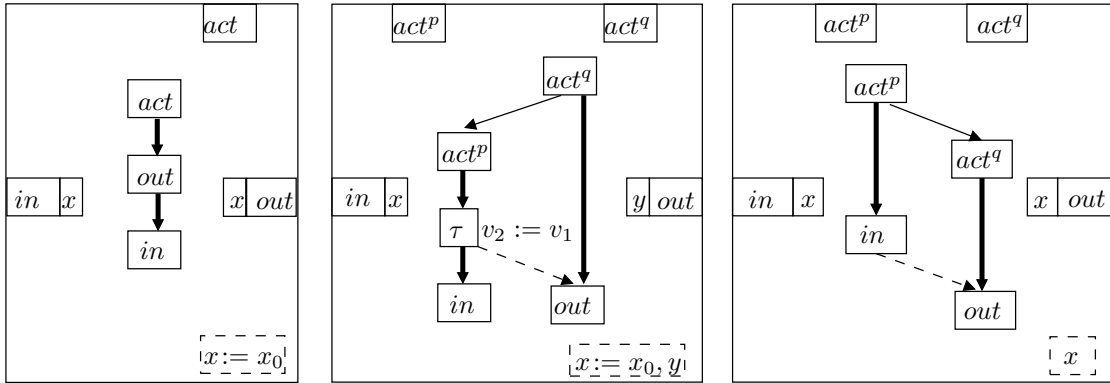


Figure 5.21: Atomic synchronous BIP components for the unit-delay Simulink block with identical sample times (left), with different sample times (middle) and the zero-order hold with different incoming and outgoing sample times (right)

We remind that these blocks can be used in Simulink to change the sample time of the incoming signal (see Chapter 3). We provide two alternative translations. The first corresponds to identical (unchanged) sample time. In this case, the modal flow graphs are rooted by a unique control BIP port *act* which triggers both the input *in* and the output *out* data BIP ports. The

second corresponds to different sample times for the incoming and the outgoing signals. In this case the input *in* and the output *out* data BIP ports are triggered by different control BIP ports act^p and act^q respectively. Moreover, the two control BIP ports are also weakly dependent in some order, and this dependency enforces the Simulink restriction that unit-delay (resp. zero-order hold) elements can be used to increase (resp. decrease) the sample time of the signal. Furthermore, input and output data BIP ports are conditionally dependent on each other, in order to represent the expected behavior i.e., unit-delay is delaying any input for at least one (input) sample time period.

The following theorem is a consequence of modularity of the translation and gives important structural properties.

Theorem 6 *A synchronous BIP component M_B obtained by the translation of a Simulink model B that is built according to the restrictions for simulating in “fixed-step solver in single-tasking mode”, enjoys the following structural properties:*

- *is well-triggered;*
- *every data BIP port is strongly dependent on exactly one of the control BIP ports;*
- *is confluent and deadlock-free.*

5.2.3 Translation of Triggered Subsystems

Triggered subsystems are translated into synchronous BIP components with a unique control BIP port act^\perp and several input $\{in_1, in_2, \dots, in_i\}$ and output $\{out_1, out_2, \dots, out_j\}$ data BIP ports, one for every Simulink inport and outport respectively defined within the Simulink model.

The general interfaces of synchronous BIP component that represent triggered subsystem is shown in Figure 5.22.

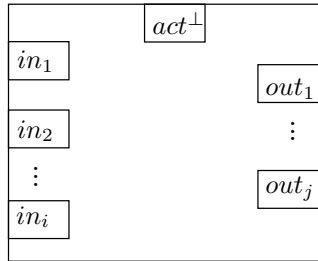


Figure 5.22: The general interface of synchronous BIP component representing triggered Simulink subsystems

According to Simulink restrictions, all atomic blocks within a triggered subsystem have inherited sample time. Moreover, a triggered subsystem can only contain basic Simulink blocks, other triggered subsystems but not enabled and continuous time general subsystems. Hence the only possible connections within a triggered subsystem are dataflow links between different blocks and ports and triggering links which activate inner triggered subsystems.

The translation of triggered subsystems is structural. The synchronous BIP component that represents a Simulink triggered subsystem is obtained by composition of its constituent components. The composition which is performed by the connectors, reflects the dataflow and activation links used within the triggered subsystem.

More precisely, the translation proceeds as follows:

First, there are generated synchronous BIP components that represent the constituent Simulink blocks of the triggered subsystem. We distinguish the following three categories:

- *Simulink inports and outports* are translated as shown in the previous section. BIP components associated with Simulink blocks play a particular role in the definition of the interface of the resulting composed synchronous BIP component. The *in* (resp. *out*) data BIP ports associated to BIP components that represent Simulink inports (resp. outports) will not become part of some connector within the subsystem and it will become part of the interface.
- *Simulink atomic blocks* are translated as shown in the previous section. All these blocks will lead to components with a unique activation event act^\perp . In particular, this is also the case for unit-delay and zero-order hold elements since they are activated by the unique sample time of the subsystem.
- *Simulink triggered subsystems* are translated recursively, following the same procedure. We simply rely on their interface in order to perform composition with other components.

Second, the components are composed by synchronization according to the dataflow and the triggering links in the Simulink model. The different type of Simulink links within a triggered subsystem and their translation in synchronous BIP are illustrated below. The continuous lines illustrate connections between control BIP ports and the dashed lines, connections between data BIP ports.

We distinguish basically three categories of connectors, presented in the following paragraphs.

Case 1

Dataflow links between blocks operating on the same sample time, e.g., Simulink output x of block A is connected to Simulink input y of block B as shown in figure 5.23.

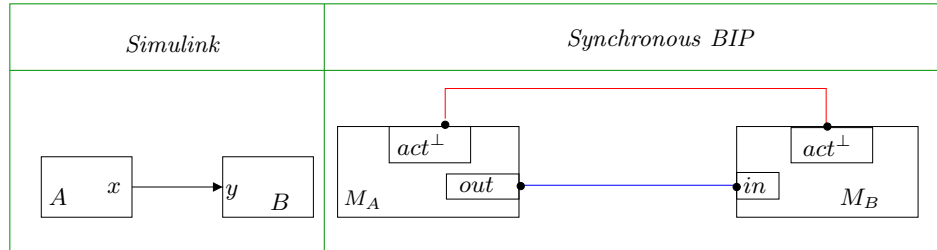


Figure 5.23: Translation of *dataflow* links between blocks operating on the same sample time within triggered subsystems

In this case, the dataflow link is translated into a strong synchronization between the *out* data BIP port of the synchronous BIP component M_A and the *in* data BIP port of the component M_B . Moreover, the control data ports of M_A and M_B are also strongly synchronized. Note that M_A and M_B are the synchronous BIP components that represent the Simulink block A and B .

Case 2

Dataflow links between blocks operating on different sample times e.g., Simulink output x of block A is connected to Simulink input y of block B which is triggered by some other event, as shown in figure 5.25.

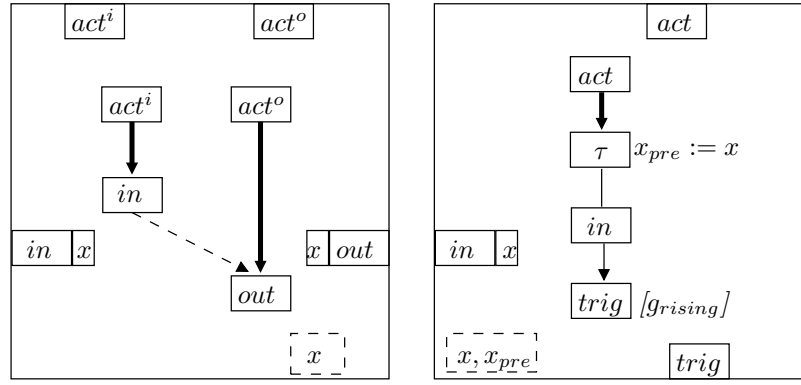


Figure 5.24: Additional components for the *sample-time-adapter* (STA) (left) and the *trigger generator* (TG) (right)

In this case, the connection is realized by passing through a *sampling-time-adapter* (STA) component which is shown in figure 5.24 (left). This component allows the correct transfer of data between a producer and a consumer activated by different events. The two control BIP ports of the *sample-time-adapter* component are synchronized with the control BIP ports of M_A and M_B respectively.

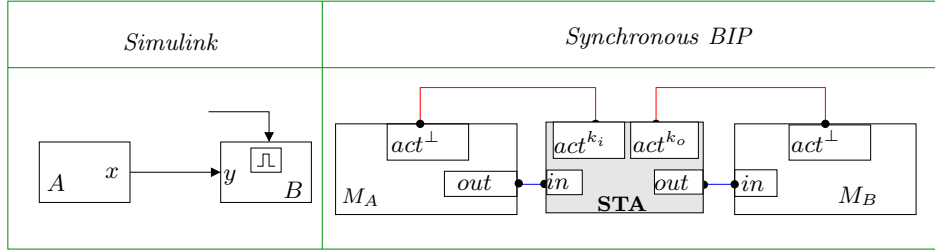


Figure 5.25: Translation of *dataflow* links between blocks operating on different sample times within triggered subsystems

Case 3

Triggering link i.e., activation of an inner triggered subsystem e.g., Simulink output x of block A is used to trigger the block B as shown in figure 5.26. In this case, the connection is realized by passing through a *trigger-generator* (TG) component which is shown in figure 5.24 (right). This component produces a triggering event *trig* whenever some condition on the input signal x holds. In Simulink this condition can be either *rising* (value changed from a negative to a positive value, $g_{rising} \equiv v_{pre} \leq 0, v_{pre} < v, 0 \leq v$), *falling* (conversely, value changed from a positive to a negative value) or either (rising or falling).

Finally, all the act^{\perp} control BIP ports which are not explicitly synchronized with a *trig* control BIP port (i.e., occurring at top level) are synchronized and exported as the act^{\perp} control BIP port of the composed synchronous BIP component.

Example 30 The Simulink model of Figure 3.14 is translated in Simulink BIP as shown in Figure 5.27. The Simulink blocks *Sine Wave*, *Trigger Signal*, *To workspace* are translated to the corresponding synchronous BIP components. Each of these components has a unique activation

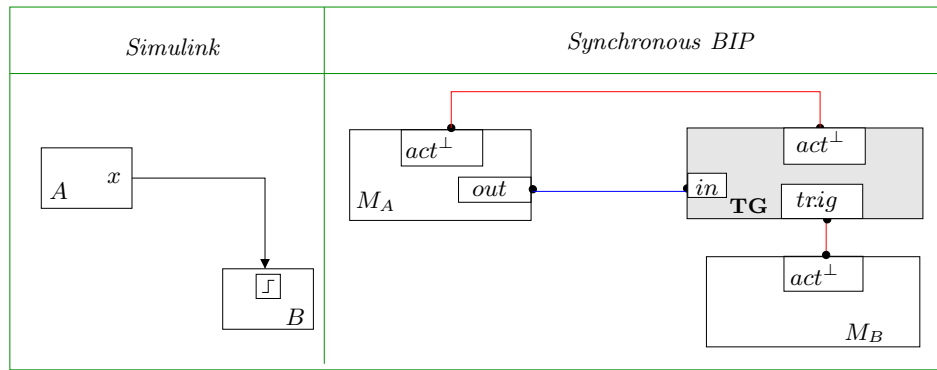


Figure 5.26: Translation of triggering links with triggered subsystems

control port act^{T_s} that corresponds to the sample time T_s of the Simulink blocks. The synchronous BIP composite component that represents the Triggered Subsystem is obtained by composition of its constituent components. That is, the synchronous BIP components that correspond to the In_1 and Out_1 Simulink ports and the UnitDelay atomic block. The Trigger Subsystem synchronous BIP component has a unique control port act^\perp . The connections between the Triggered Subsystems, the Sine Wave and the To Workspace is realized through two sampling-time-adapters (STA) respectively. The connection between the Trigger Signal and the Triggered Subsystem is realized by passing through a trigger-generator (TG) synchronous BIP component. All act^{T_s} ports are strongly synchronized and exported to the composite component.

5.2.4 Translation of Enabled Subsystems

The construction of such a subsystem is structural and incremental, extending the method described previously for triggered subsystems. As before, first there are collected the components for all the constituent blocks and then there are composed according to dataflow, triggering and enabling links defined in Simulink.

The general interfaces of synchronous BIP component that represent enabled subsystem is shown in Figure 5.28.

The translation of the new categories of Simulink links occurring in the context of an enabled subsystem is illustrated in Figure 5.29 and 5.30. The continuous lines illustrate connections between control BIP ports and the dashed lines, connections between data BIP ports.

We distinguish two new categories of connections as shown below.

Case 1

Dataflow links between subsystems having different enabling conditions e.g. Simulink outputport x of A connected to Simulink inputport y of B as illustrated in Figure 5.29.

In this case the connection in BIP is realized by passing through a *sample-time-adapter* (STA) component in order to adapt for the possible different activation times for input and output events. Only the control BIP ports act^{k_o} and act^{k_i} triggering respectively the events out^x in M_A and in^y in M_B have to be synchronized with the STA, whereas all other act control BIP ports remain unconstrained.

Case 2

enabling link i.e., conditional execution of the subsystem depending on some condition e.g.,

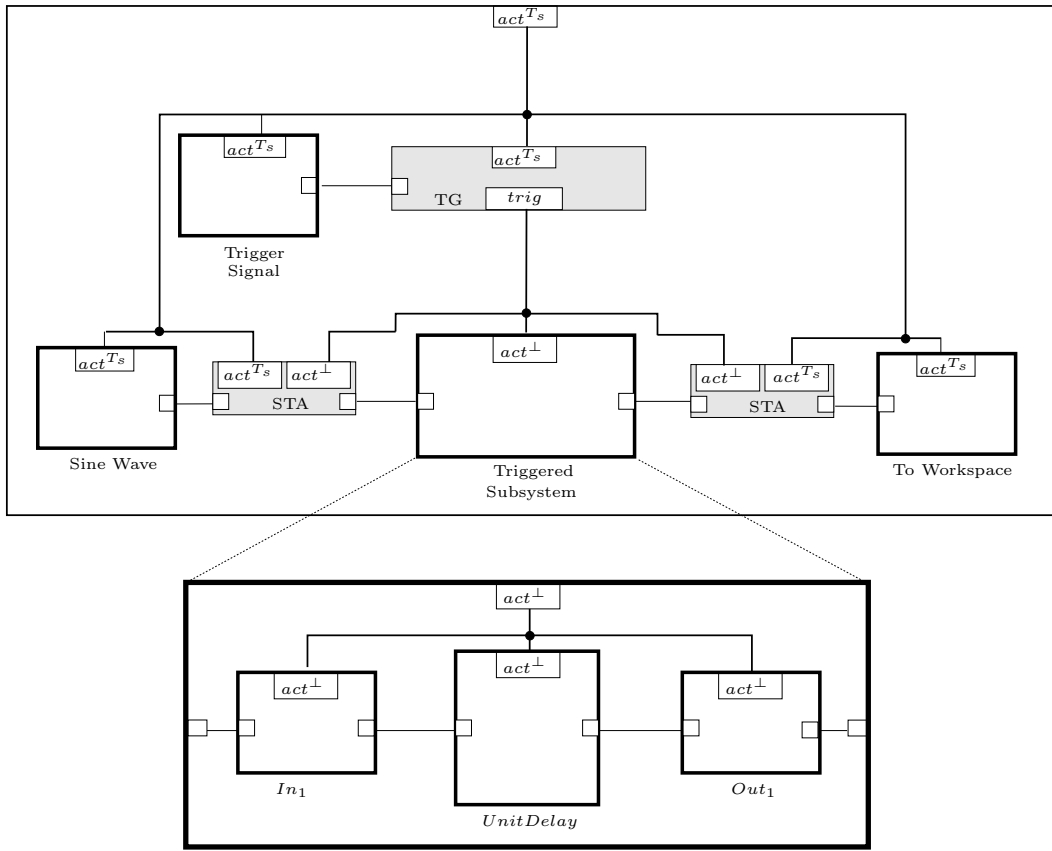


Figure 5.27: Translation of the Simulink model of Figure 3.14.

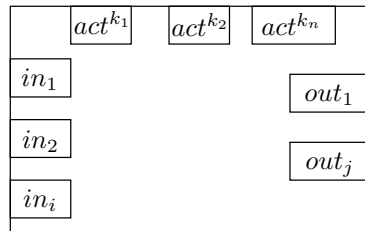
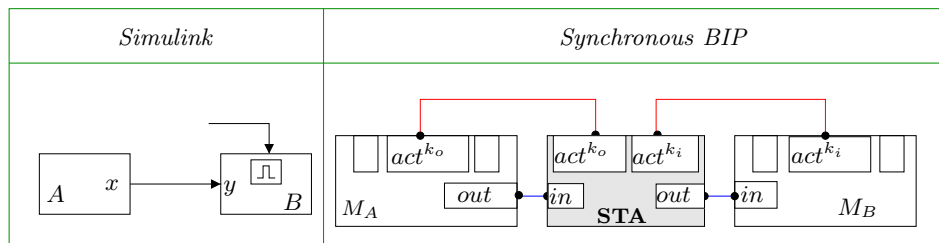


Figure 5.28: The general interface of synchronous BIP component representing enabled Simulink subsystem

Figure 5.29: Translation of *dataflow* links between subsystems having different enabling conditions

outport x of A defines the enabling condition for B as illustrated in figure 5.30.

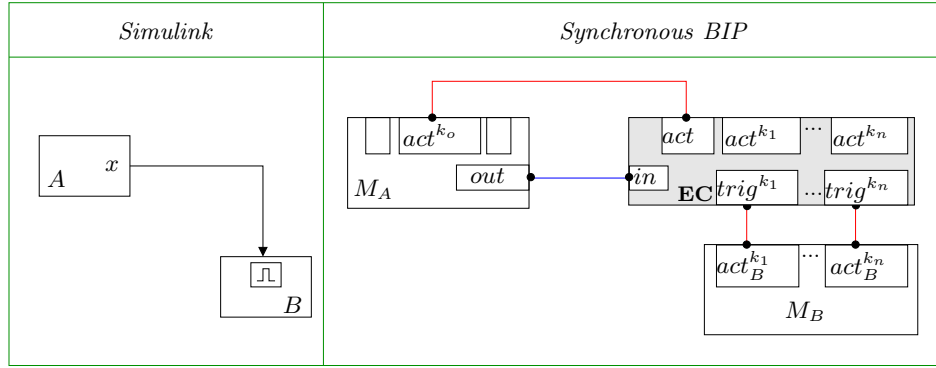


Figure 5.30: Translation of *enabling* links in enabled subsystem

Such a connection requires an additional component called *enabling condition* (EC) illustrated in Figure 5.32. Such a component filters out any (periodic) control BIP port act^{k_i} occurring when the input signal x is false or negative. Otherwise it propagates the event corresponding to the control BIP port act^{k_i} renamed as $trig^{k_i}$.

Any other category of connections is handled as for triggered subsystems.

Finally, all activation events act^{k_i} which correspond to the same sample time k_i and which are not explicitly synchronized with a $trig^{k_i}$ control BIP port, are strongly synchronized and exported as the act^{k_i} control BIP port on the interface of the composed synchronous BIP component.

Example 31 The Simulink model of Figure 3.16 is translated in Simulink BIP as shown in Figure 5.31.

The Simulink blocks are translated to the corresponding synchronous BIP components. The Subsystem is translated recursively to a composite component. The necessary smalling-time-adapters (STA) and the enabling condition (EC) are added for communication between the synchronous BIP components. There are produced four different activation ports one for each of the different sample times of the Simulink model.

5.2.5 Clock Generator

A clock component Clk_B produces the activation events that correspond to the different sample times occurring in the model. The activation events of the Clk_B component are produced using a global time reference and obey the corresponding ratio, respectively k_1, k_2, \dots . A concrete example of such a Synchronous BIP component is provided in Figure 5.33. This component generated the activation events for the example of figure 5.31. The same construction can be easily generalized to any number of sample times.

Example 32 Figure 5.33 shows the Synchronous BIP component that produces different clock events for every 12.5, 25, 37.5 and 50 units of time.

The component uses a variable c to measure time and has six ports $tick$, $act^{12.5}$, act^{25} , $act^{37.5}$ and act^{50} . The port $tick$ represents a global clock tick. This port is triggered every synchronous step and increases the value of c by 12.5. A clock event $(act^k)_{k=12.5,25,37.5,50}$ is then produced each time the period k divides the current time c , denoted by $k|c$. The port $reset$ is used to reset c every 150 time units, that is, the least common multiple of all the periods. The guards and the

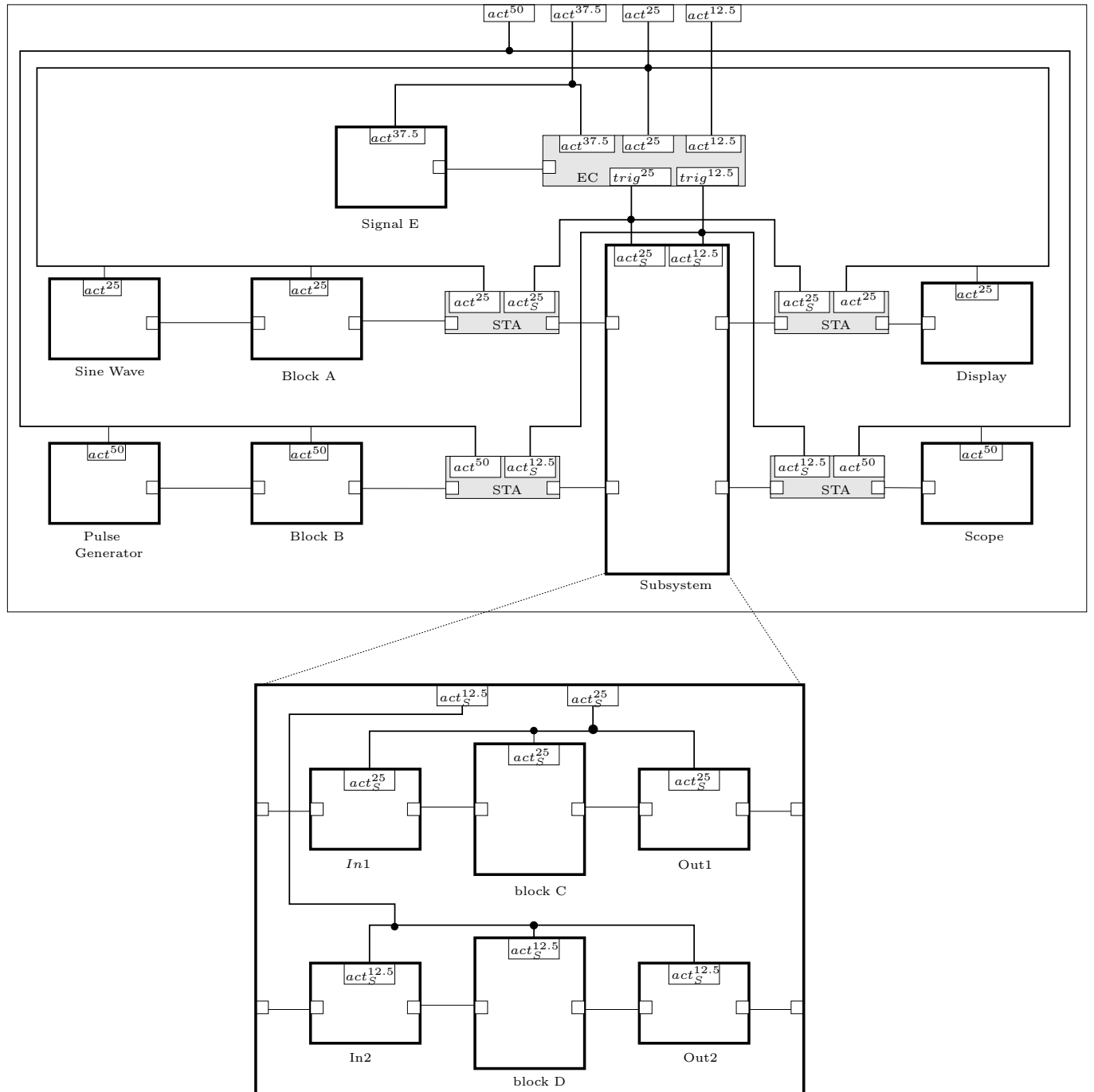


Figure 5.31: Translation of the Simulink Subsystem of Figure 3.16.

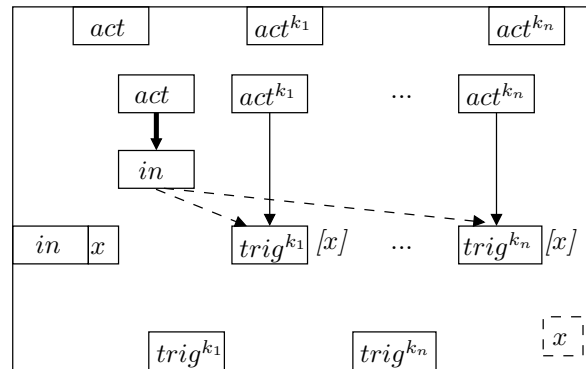
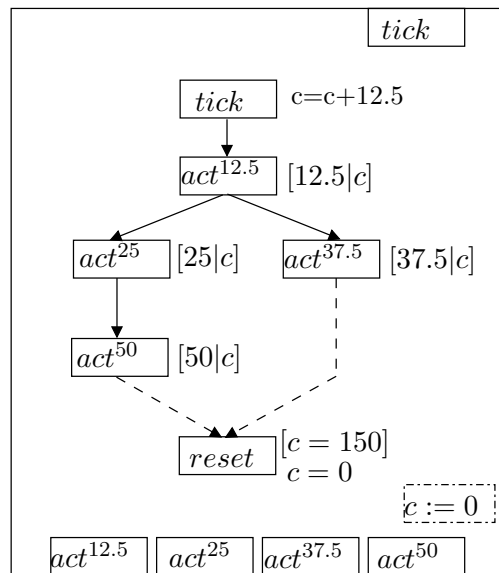
Figure 5.32: The *enabling condition*(EC) component

Figure 5.33: A clock generator component

causal dependencies ensure that, in every synchronous step, exactly one of the following sequences is executed: $tick$, $tick \cdot act^{12.5}$, $tick \cdot act^{12.5} \cdot act^{25}$, $tick \cdot act^{12.5} \cdot act^{37.5}$, $tick \cdot act^{12.5} \cdot act^{25} \cdot act^{50}$, $tick \cdot act^{12.5} ((act^{25} \cdot act^{50}) | act^{37.5}) \cdot reset$ (where $|$ denote the shuffling of two sequences).

5.2.6 Translation of a Simulink Model

The complete translation of a Simulink block B is carried out by strongly synchronizing the clock Clk_B component with the Synchronous BIP component M_B as shown in Figure 5.34. The activation ports of the Clk_B component are strongly synchronized with the ports of the component M_B that correspond to the same sample time.

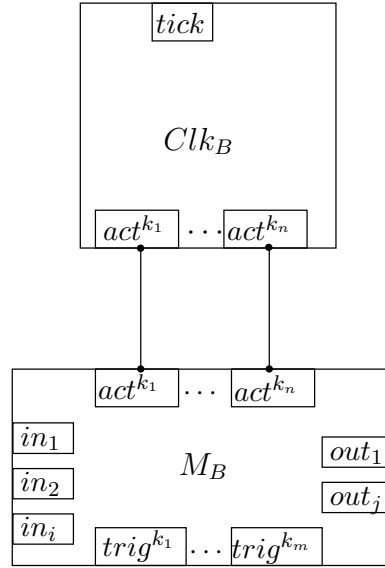


Figure 5.34: Complete translation of a Simulink model

5.2.7 Implementation of the Translation

The translation from MATLAB/Simulink to Synchronous BIP has been implemented in the `Simulink2S-BIP` tool. It parses MATLAB/Simulink model files (`.mdl`), and produces Synchronous BIP models (`.bip`). The generated models reuse a (hand-written) predefined component library of atomic components and connectors (`simulink.bip`). This library contains the most common atomic blocks (sources, combinatorial operators, memories, transfer functions, etc) as well as the most useful connectors (for in/out data transfer and for control activation). Chapter 6 presents a list with Simulink models that we translated into Synchronous BIP.

5.2.8 Similar Translations

The work in [7] presents a translation for a subset of MATLAB/Simulink and Stateflow into equivalent hybrid automata. The translation is specified and implemented using a metamodel-based graph transformation tool. The translation allows semantics interoperability between the Simulink's standard tools and other verification tools.

The work of [65, 70] is probably the closest to our work. These papers present a compositional translation for discrete-time Simulink and respectively discrete-time Stateflow models into Lustre programs [43]. This work leverages the use of validation and (certified) code generation

techniques available for Lustre to Simulink models. The translation consists of three steps: type inference, clock inference, and hierarchical bottom-up translation. It has been implemented by the S2L tool [3].

We can also mention [57] where a restricted subset of MATLAB/Simulink, consisting of both discrete and continuous blocks, is translated into the COMDES framework (*Component-based Design of Software for Distributed Embedded Systems*). However, this work focuses on the relation between control engineering and software engineering related activities.

Finally, [58] presents a tool which automatically translates discrete-time Simulink models into the input language of the NuSMV model checker. This translation allows efficient symbolic verification techniques to Simulink models used in safety-critical systems.

The fragments translated in [7], [57] and [58] are either incomparable or handled differently. For instance, the translation reported in [7] focuses on continuous-time models, and allows for a limited discrete behavior represented using switches. The work [58] covers an important part of the discrete-time fragment, and in particular, n -dimension signals and related operators (mux, demux). Nevertheless, it does not consider blocks such as the discrete transfer functions, and moreover, it seems to be restricted to models with unique sample time. The solution chosen in [57] for handling multiple sample times is also different. Although, the precise translation is not explained thoroughly in the paper, it is claimed that it relaxes the exact timing constraints of Simulink, since they are fundamentally impossible to implement and unnecessarily restrictive.

We cover almost the same discrete-time fragment as [70]. Also, we adopt exactly the same semantics choices. However, we believe that our translation method provides a much understandable representation, which better illustrates the control and data dependencies in the Simulink model. For example, we are using (generic) explicit components for adaptation of sample times for signals going into/coming from subsystems. In the Lustre translation, this adaptation is hard-coded using sampling/interpolation operators and gets mixed with other (functional) equations of the subsystem. Furthermore, we do not hard-wire the sample time of signals using absolute clocks. Instead, we merely track all the sample time dependencies (e.g., equalities) within the model and define them only once, at the upper layer, using a sample-time period generator.

Finally, as a general remark, our models have a graphical representation that is closest to the Simulink models. There is almost an one to one mapping of each Simulink block to a Synchronous BIP component. In that sense, the produced Synchronous BIP model can be easily understood by Simulink users.

5.3 Conclusion

This chapter presented transformations of synchronous formalisms into Synchronous BIP. We proposed transformations of LUSTRE and discrete MATLAB/Simulink into well-triggered synchronous systems. The translations are modular and exhibit data-flow connections between models within heterogeneous BIP designs. Moreover, they enable the application of validation and automatic implementation techniques already available for BIP. Both translations have been implemented to tools.

The principles of the above transformations can be applied to other synchronous formalisms too. Till now, we have been experimented with two more formalism, Scilab [4] and StreamIT [5].

In next chapter, we will generate C code from Synchronous BIP components corresponding to LUSTRE nodes and Simulink models. We will use this code to compare performances with the C code generated from LUSTRE and Simulink respectively.

Chapter 6

Code Generation for Synchronous BIP

This chapter presents two implementations for generating C code from Synchronous BIP models, sequential implementation and distributed implementation. The *sequential* implementation generates a single endless loop. The *distributed* implementation transforms modal flow graphs to a particular class of Petri nets that can be mapped to Kahn process networks.

The chapter is structured as follows. Section 6.1 describes the sequential implementation. It presents the main algorithm for the generation of C code and some experimental results on LUSTRE and MATLAB/Simulink examples. Section 6.2 describes two methods for the distributed implementation. Both methods use transformations to Petri nets which can be mapped to Kahn process Networks. Section 6.4 draws some conclusions.

6.1 Sequential Implementation

This section presents the sequential implementation of Synchronous BIP models. The code generator takes as input a Synchronous BIP compound component which is the set of atomic components connected through interactions. The code generator produces single endless loop C code. The execution of one *loop* cycle corresponds to the behavior of the Synchronous BIP model in one computational step. The code generator for the sequential implementation has the following algorithm:

1. *Static composition of the compound component to an atomic component:* Composition of Synchronous BIP atomic components is defined as a partial internal operation parametrized by a set of interactions. Given a set of Synchronous BIP atomic components, we get a product component by composing their modal flow graphs. The produced component consists of a set of ports that correspond to the set of interactions between atomic components. Dependencies between ports are inherited from atomic components (see also Definition 17, Chapter 4). We assume that the produced Synchronous BIP atomic component satisfies two important properties. First, it is *acyclic*, that is, the set of dependencies do not produce a closed walk. Second, it is *well-triggered* that is ports have exclusively either strong or weak causes. Moreover, for each port there exists a unique minimal strong cause.
2. *Find an execution order for all ports of an atomic component:* Given the atomic component produced in the previous step, we determine the order of execution of ports of the

component. The order is computed by applying a topological sorting algorithm. Causal dependencies enforce source ports to be executed before the target ports.

3. *Generation of the code:* The code we generate from a Synchronous BIP component is a single loop C code. Inside the loop, all ports of the component are executed in the order defined by the topological sorting. The execution of a port p , is marked with an associated boolean variable `exec_p`. A port p is then executed if:

- its assigned guard g_p has been evaluated to *true*;
- all its strong and weak predecessor ports p_1, \dots, p_n have already been executed, i.e. the boolean variables $exec_p_1, \dots, exec_p_n$ are *true*.
- if conditional predecessors are present, they have already been executed, enforced by the execution order of the ports.

The following block of code shows the execution of a port p inside the loop of the generated C code for an atomic Synchronous BIP atomic component. The execution of the port p is followed by the computation of its assigned function f_p . Moreover, its associated boolean variable `exec_p` is set to *true*.

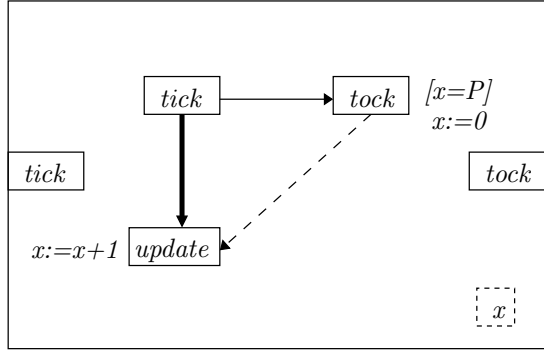
```
if (g_p ∧ (exec_p1 ∧ exec_p2 ∧ ... ∧ exec_pn)) {
    f_p;
    exec_p = true;
}
```

Example 33 *Figure 6.1 (left) shows the generated C code for the Tick-tock example (Figure 6.1 (right)).*

Boolean variables `exec_tick`, `exec_tock` and `exec_update` are assigned to the ports of the component tick, tock and update. Initially, they are all set to *true*. Ports are executed in the order tick, tock, update. Port tick has no predecessors, so execution or not execution of this port depends on its associated guard which is *true*. We mark the execution of the tick port by assigning to *true* the variable `exec_tick`. Port tock can be executed if its associated guard is *true* and moreover its predecessor port has been executed, i.e. `exec_tick` is *true*. Execution of tock is followed by assigning to zero the variable x and to *true* its associated variable `exec_tock`. Finally, execution of port update depends on its associated guard g_{update} which is always *true* and on whether its predecessor port has been executed, i.e. `exec_tick` is *true*. The execution of update is followed by increasing x by one and assigning its boolean value `exec_update` to *true*.

6.1.1 Experimental Results

The code generator for producing C code from Synchronous BIP models has been implemented to the S-BIP2C tool. It has been implemented in Java and has 5.000 lines of Java code, excluding the auto generated files. This section presents experimental results on the sequential implementation for Synchronous BIP models. We generate C code for Synchronous BIP components that correspond to LUSTRE and MATLAB/Simulink models using the *S-BIP2C* tool. We compare performances with the C code generated from the *lustre2C* code generator and the *Real-Time Workshop* for MATLAB/Simulink.



```

while(true) {
    bool exec_tick = false;
    bool exec_tock = false;
    bool exec_update = false;

    /* execution of tick */
    if (true) {
        exec_tick:=true;
    }

    /* execution of tock */
    if ( (x == P) && exec_tick) {
        x = 0;
        exec_tock:=true;
    }

    /* execution of update */
    if (true && exec_tock) {
        x = x + 1;
        exec_update:=true;
    }
}

```

Figure 6.1: Generated C code (right) for the Tick-tock example (left)

Example	#SC	#MC	t_{lus}	t_{s-bip}	t_{bip}
<i>watchdog</i>	8	0	1,7	1,5	969,5
<i>mux</i>	6	6	1,9	1,4	843,6
<i>async</i>	6	3	1,2	1,8	936,1

Figure 6.2: Experimental results for LUSTRE examples

Results on LUSTRE Models

Table 6.2 summarizes experimental results on several LUSTRE models. The table provides information about the complexity of these models. $\#SC$ is the number of single-clock components and $\#MC$ is the number of multi-clock components. For all examples we have produced executable code using respectively `lustre2C` code generator, the `S-BIP2C` code generator and the `BIP2C` code generator (code generator for BIP models). Table 6.2 reports the execution times measured using the three implementations (i.e. columns t_{lus} for `lustre2C`, t_{s-bip} for `S-BIP2C` and t_{bip} for `BIP2C`) for 10^7 iterations. For these examples, the experiments show comparable execution time between the C code produced by the Synchronous BIP code generator and the flat C code produced by the LUSTRE code generator. Moreover, the C code produced by the BIP generator and executed by the BIP engine has a clear overhead 600:1 compared to sequential C code produced by the Synchronous BIP. Table 6.2 summarizes experimental results on several LUSTRE models.

Results on MATLAB/Simulink Models

Table 6.3 summarizes experimental results on several MATLAB/Simulink models.

Ex.	#A	#P	#T	#E	n	t_{rtw}	t_{s-bip}
<i>64-bit counter</i>	365	0	60	0	10^6	3,330s	1,863
					10^7	59,283s	25,953s
<i>Anti-lock breaking</i>	39	2	0	0	10^4	0,017s	0,016s
					10^6	0,317s	1,273s
<i>Steering Wheel</i>	120	15	1	0	10^6	1,863s	3,330s
					10^7	7,221s	31,899s
<i>Enabled Subsystem</i>	24	0	0	2	10^6	0,382s	0,196s
					10^7	3,201s	1,756s
<i>Thermal model house</i>	45	3	0	2	10^6	0,562s	0,751s
					10^7	5,215s	7,565s

Figure 6.3: Experimental results for MATLAB/Simulink examples

We have discretized and translated several demo examples available in MATLAB/Simulink including the *Anti-lock Breaking* system, the *Enabled subsystem demonstration* and the *Thermal model of a house*. We have also translated examples provided in [70] like the *Steering Wheel* application. Finally, we have considered several artificial benchmarks like the *64-bit counter*. The table provides information about the complexity of these models. #A is the number of atomic blocks, #P the number of periodic blocks, #T the number of triggered subsystems and #E the number of enabled subsystems. As illustrated in the table, our translation tool actually covers a significant number of Simulink concepts.

In all cases, the simulation traces produced respectively by Simulink in simulation mode and by Synchronous BIP are almost identical. We have observed few small differences for some examples, which are due to a different representation of floating-point numbers in the Simulink and Synchronous BIP.

Finally, for all examples we have produced executable code using respectively the *Real-Time Workshop* tool of MATLAB for generating C code and the *S-BIP2C* code generator. Table 6.3 reports the execution times measured using the two implementations (i.e., columns t_{rtw} for Real-Time Workshop, t_{s-bip} for Synchronous BIP) for different numbers of iterations n . We observe that the Synchronous BIP generated code is comparable to the Real-Time Workshop in almost all the considered examples. The results are measured on a standard PC Linux machine. Nevertheless, they provide only a preliminary and partial comparison since (1) our translation does not (yet) cover all the models that can be actually handled by the Real-Time Workshop and (2) the two code generators do not necessarily target the same execution platforms.

6.2 Distributed Implementation

This section presents two methods for generating distributed code for Synchronous BIP components, the *direct* method and the *cluster-oriented* method. Both methods propose a representation of Synchronous BIP components to a particular class of Petri nets which can be mapped to Kahn process networks (KPN) [49].

KPN is a model of computation (MoC) for modeling distributed systems. Kahn process networks are directed graphs where nodes represent *processes* and arcs represent *channels* of communication between processes. Channels are infinite FIFO queues. Writing to a channel is non blocking but reading can be blocking. If a process tries to read from an empty input it is

suspended until it has enough input data and the execution is switched to another process.

Processes of KPN are deterministic. For the same input they always produce exactly the same output. Moreover, each process of a KPN has a sequential behavior. It consumes data from its input FIFO queues and produces tokens to the output queues. Finally each FIFO queue has one source and one destination. One way to describe the semantics of processes of a KPN is using *Petri Nets*. Processes are mapped to transitions and FIFO queues to places.

Before presenting the two methods for generating distributed code from Synchronous BIP components, we give the following definitions.

Definition 20 (Cluster) *Let $M = (X, P, D)$ a modal flow graph with X the set of variables, P the set of ports and D the set of dependencies such that $D = D_s \cup D_w \cup D_c$. M can be decomposed in clusters C_i with $i \in I$ such that:*

- $C_i \subseteq P$, $\forall i \in I$ and $\bigcup_{i \in I} C_i = P$, i.e., the set of ports P forms clusters C_i ;
- $C_i \cap C_j = \emptyset$ with $i \neq j$, i.e., each port belongs to exactly one cluster;
- For $p \in C_i$ and $q \in C_j$ with $p \xrightarrow{s} q$ then $C_i = C_j$ i.e., strong dependencies are allowed only within clusters;

The methods we present in this section can be applied to Synchronous BIP components which satisfy the following properties:

1. Each cluster C_i is triggered from at most an other cluster C_j . That is, the root of a cluster C_i has at most one weak dependency.
2. Conditional dependencies between two clusters exist if only their predecessors are also weakly dependent.
3. For the sake of simplicity, we restrict to clusters that have sequential behavior, that is, every port has precisely one successor port (except for the final one).

Example 34 *Figure 6.4 shows a Synchronous BIP component which consists of three clusters C_1, C_2 and C_3 rooted by b, a and f respectively. The component has the properties we described above. Each cluster has only strong dependencies between ports. For clusters C_1 and C_3 there is a unique weak dependency that triggers their root. Moreover, conditional dependent ports have predecessors which are weakly dependent. Finally, all three clusters have sequential behavior, that is, each port has a unique predecessor and successor.*

6.2.1 Direct Method for Distributed Code Generation

The “direct” method for generation of distributed code from Synchronous BIP components was proposed by Goesller and Smeding [68]. This method consists of two steps:

1. *Transformation of modal flow graphs to Petri nets.* The procedure of this transformation is the following:
 - Each port p of a modal flow graph is mapped to a pair of transitions, a positive t_p and a negative $t_{\bar{p}}$. For a weak dependency, an additional negative transition $t_{\hat{p}}$ is added. For weak dependent ports p and q such that $p \xrightarrow{w} q$, there are the possible executions: 1) both p and q are executed, mapped to transitions t_p and t_q respectively, 2) none of them is executed, mapped to transitions $t_{\bar{p}}$ and $t_{\bar{q}}$ respectively and 3) port p is executed but port q is not executed, mapped to transitions t_p and $t_{\hat{q}}$ respectively. A global transition t_{sync} is added to denote the termination of the execution of the component.

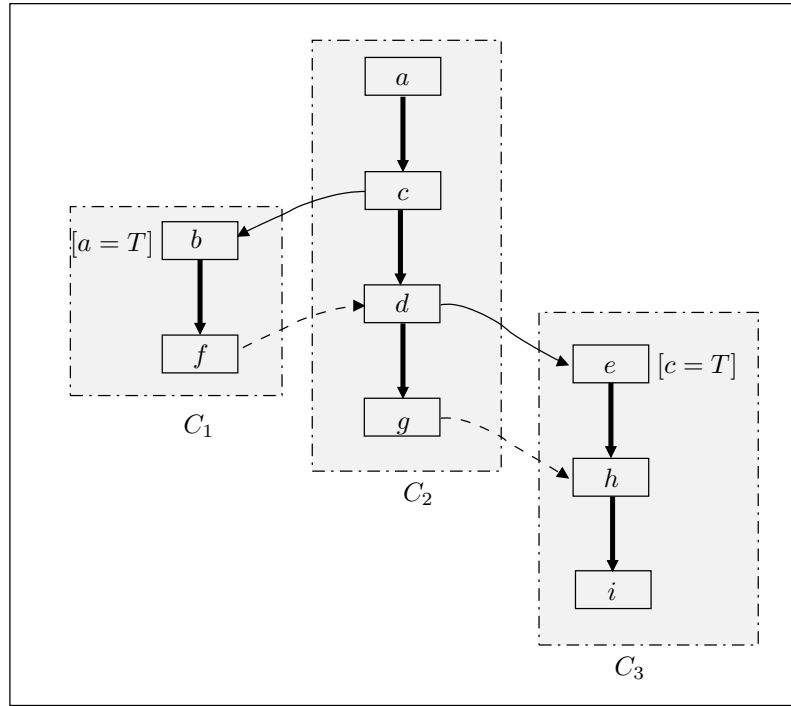


Figure 6.4: A Synchronous BIP component formed in clusters C_1 , C_2 and C_3

- Dependencies are mapped to places. Each strong and weak dependency is mapped to two places corresponding to execution or not execution of the cause of the dependency. Conditional dependency is mapped to a unique place since a dependent port can be executed even if its cause is not executed.
2. *Mapping the constructed Petri nets to Kahn Process Networks.* Transitions are mapped to processes and places to FIFO communication channels as explained below:
- Each pair or triple of transitions $t_p, t_{\bar{p}}$ and $t_{\hat{p}}$, corresponding to a port p , is a *process* in the KPN
 - Places l_{pq} and $l_{\bar{p}q}$ (if any) as well as places corresponding to the minimal and maximal causes, represent FIFO buffers. These are channels for communication between interconnected processes.

The Petri net transformation of a modal flow graph for the “direct” method is a BIP component which is defined as shown in Definition 21. We remind that $\min P$ defines the set of minimal causes, that is, ports with no dependencies. This set is defined formally as $\min P = \{q \mid \neg \exists p.p \rightsquigarrow q\}$. We define $\max P$ the set of maximal ports, that is ports which do not trigger the execution of any other port. That is, $\max P = \{q \mid \neg \exists p.p \rightsquigarrow q\}$.

Definition 21 *Let a synchronous BIP component $B^f = (X, P, D)$. We define a BIP component (X, P, N) such that:*

- X is the set of variables
- P is the set of ports; moreover for each port $p \in P$ the associated set of exported variables is X_p .

- the Petri net is defined as $N = (L, T, F, L_0)$, where:

- the set of places L such that

$$L = \{l_{pq}, l_{\bar{p}q} \mid p \xrightarrow{s,w} q\} \cup \\ \{l_{pq} \mid p \xrightarrow{c} q\} \cup \\ \{l_p \mid p \in \min P\} \cup \\ \{l_q \mid q \in \max P\}$$

- the set of transitions T such that

$$T = \{t_q, t_{\bar{q}} \mid p \xrightarrow{s,c} q\} \cup \\ \{t_q, t_{\bar{q}}, t_{\hat{q}} \mid p \xrightarrow{w} q\} \cup \\ \{t_p, t_{\bar{p}} \mid p \in \min P\} \cup \\ t_{sync}$$

- the token flow relation $F \subset L \times T \cup T \times L$ is defined as follows:

- * for $p \xrightarrow{s} q$, add $(t_p, l_{pq}), (l_{pq}, t_q)$ and $(t_{\bar{p}}, l_{\bar{p}q}), (l_{\bar{p}q}, t_{\bar{q}})$ and $(t_{\hat{p}}, l_{\bar{p}q}), (l_{\bar{p}q}, t_{\hat{q}})$
- * for $p \xrightarrow{w} q$, add $(t_p, l_{pq}), (l_{pq}, t_q)$ and $(t_{\bar{p}}, l_{\bar{p}q}), (l_{\bar{p}q}, t_{\bar{q}})$ and $(l_{pq}, t_{\hat{q}}), (t_{\hat{p}}, l_{\bar{p}q})$
- * for $p \xrightarrow{c} q$, add $(t_p, l_{pq}), (l_{pq}, t_q)$ and $(t_{\bar{p}}, l_{pq}), (l_{pq}, t_{\bar{q}})$ and $(t_{\hat{p}}, l_{pq}), (l_{pq}, t_{\hat{q}})$
- * for $\{q \in \max P\}$, add $(t_q, l_q), (t_{\bar{q}}, l_q), (t_{\hat{q}}, l_q)$ and (l_q, t_{sync})
- * for $\{p \in \min P\}$, add (t_{sync}, l_p) and $(l_p, t_p), (l_p, t_{\bar{p}})$

- L_0 is the set of initially marked places such that $L_0 \subset L$ and $L_0 = \{p \mid p \in \min P\}$.

Figure 6.5 represents graphically the correspondence of causal dependencies for modal flow graphs to Petri nets according to the method described below.

Example 35 Figure 6.6 illustrates the representation of the Synchronous BIP component of Figure 6.4 into Petri nets as described above. Ports are mapped to sets of positive and negative transitions and interconnected through places. Positive transitions t_a, t_b, \dots, t_i correspond to the execution of the ports a, b, \dots, i respectively. Negative transitions $t_{\bar{a}}, \dots, t_{\bar{i}}$ denote that the corresponding ports are not executed in the current step. The negative transitions $t_{\hat{a}}$ correspond to weak transitions and denote not execution of the dependent ports in contrast to its causes. The cycle of an execution is completed when the t_{sync} transition is executed.

To prove the correctness of the implementation, that is, all executions of the produced Petri net conform to dependencies enforced by the modal flow graphs, we consider the following facts:

1. Execution steps are explicitly separated using an explicit synchronization, implemented by the “clk” process (corresponding to the t_{sync} transition)
2. The execution of a port is decided and realized by one process.
3. Each dependency is encoded by one FIFO channel. The data sent over the FIFO channels carry explicit information about execution or not execution of the source port.
4. Each process P_p is executed involving the following three steps:
 - (a) It reads data from all its input FIFO channels. This data informs the process about the execution status of its proceeding ports;
 - (b) Based on the information acquired from the FIFO channel and the local guard it decides about the execution of the port;

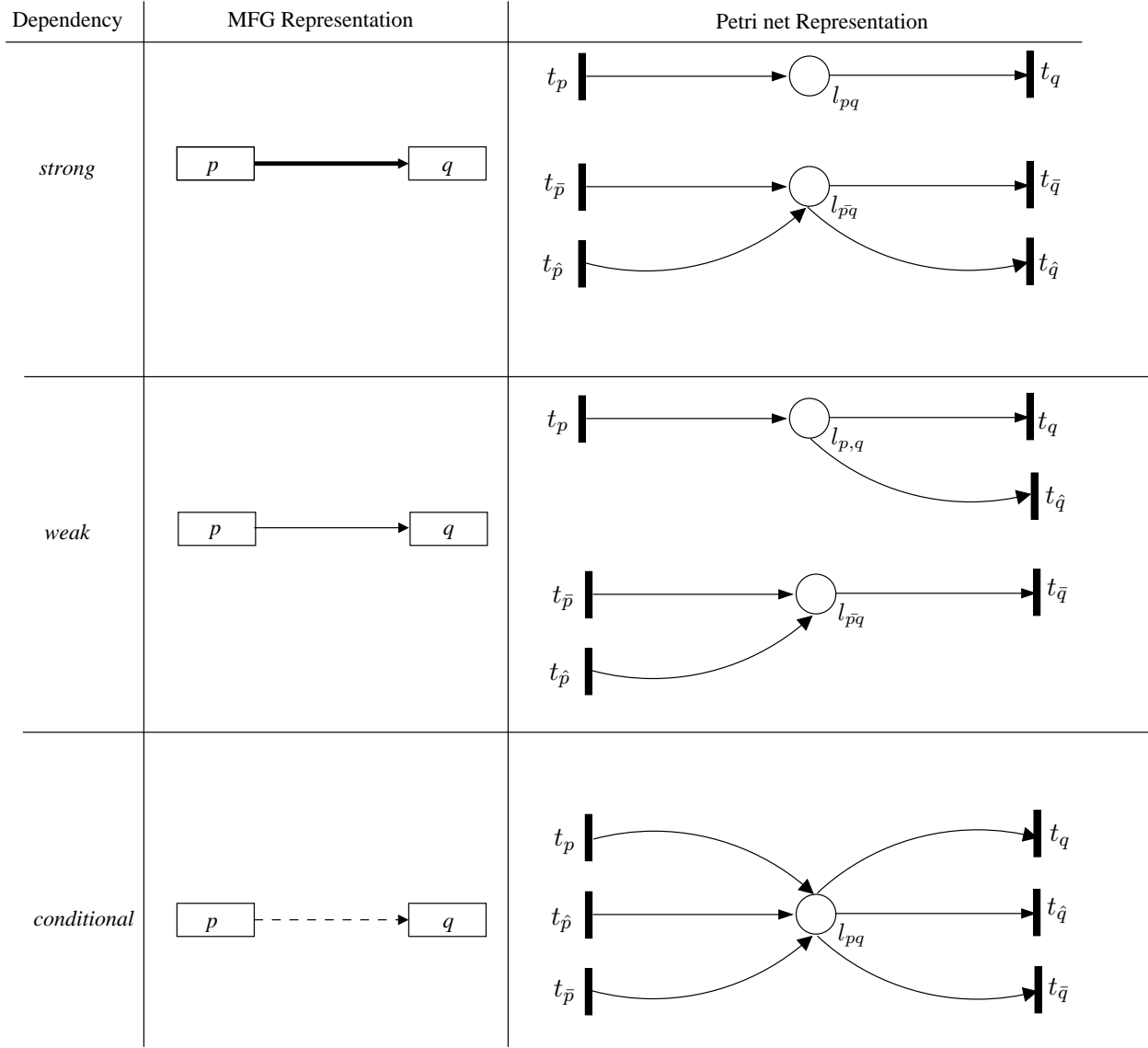


Figure 6.5: The transformation of causal dependencies for modal flow graphs to Petri nets (*Direct method*)

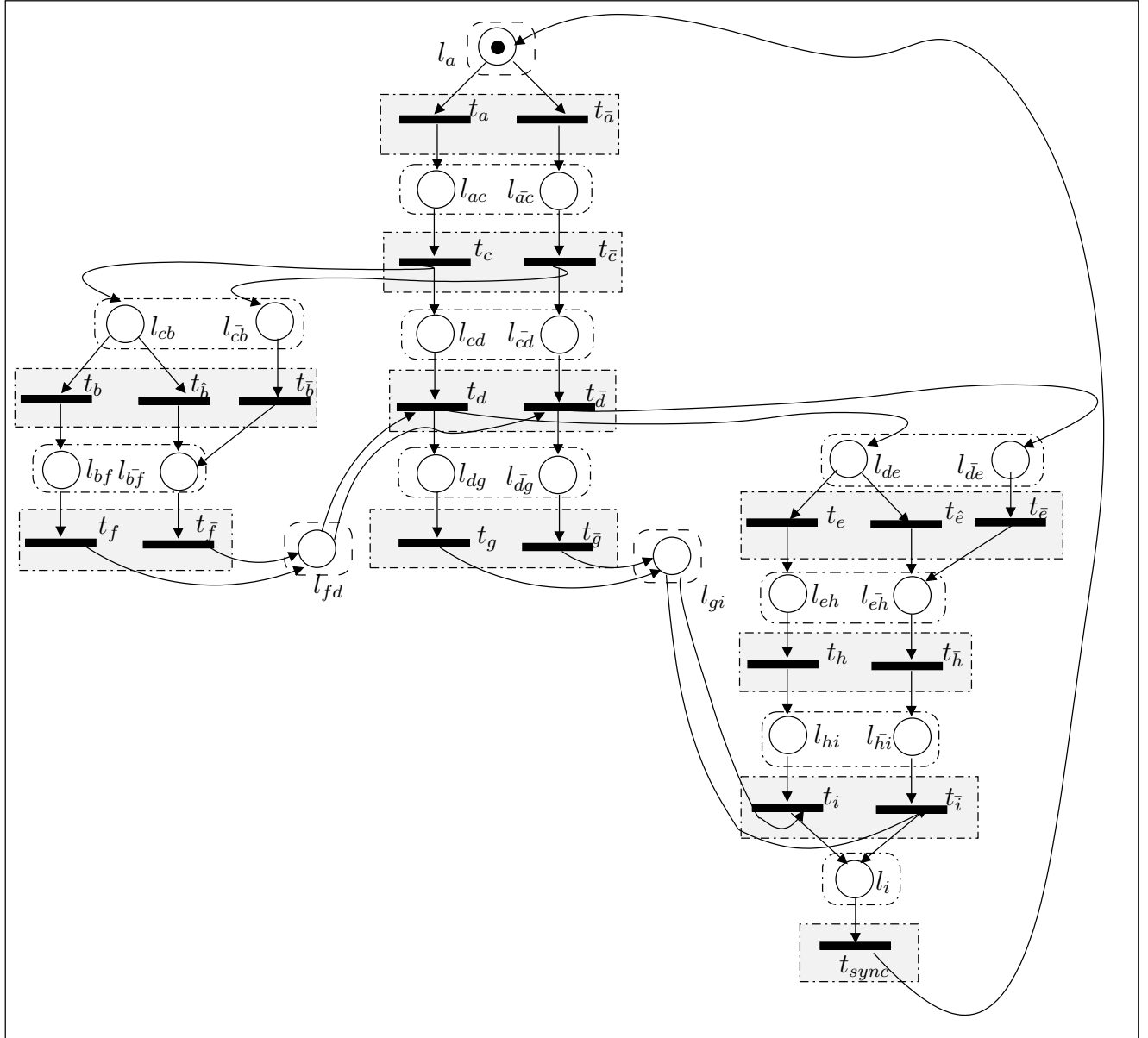


Figure 6.6: The Petri net representation (*Direct method*) for the Synchronous BIP component of Figure 6.8

- (c) It writes data to all its output FIFO channels. This data informs its dependent processes about its execution status.

Based on those observations, it holds that, during each execution step, all the dependencies between ports are explicitly examined. The execution at each step is therefore a valid execution of the modal flow graph as defined by its semantics.

6.2.2 Cluster-oriented Method for Distributed Code Generation

The “cluster-oriented” method for generation of distributed code from Synchronous BIP components considers modal flow graphs that can be formed in clusters as described in Definition 20. Clusters are formed of ports which have only strong dependencies. Communication between clusters is done through conditional and weak dependencies. The “cluster-oriented” method consists of two steps:

1. *Transformation of the modal flow graphs to Petri nets.* The procedure of the transformation is the following:
 - Each port of a cluster is mapped to as many transitions as to represent the execution or not of all predecessor weak dependencies.
 - Strong dependencies are encoded from as many places as the produced transitions. Weak and conditional transitions are encoded from one place each.
2. *Mapping the constructed Petri nets to Kahn process networks.* The produced Petri net can be seen as a set of processes communicating through FIFO channels. Each process is a set of transitions that corresponds to the the ports of each cluster of the modal flow graph. FIFO channels are the places that encode the weak and conditional dependencies.

The Petri net transformation of a modal flow graph for the “cluster-oriented” method is a BIP component which is defined as shown in Definition 22. We give the following notations:

1. For a port $p \in C_i$ we define the set Pre_p of predecessors for a port p such that $Pre(p) = \{x \in C_i \mid x \xrightarrow{s^*} p\}$;
2. For each port p' in the set $Pre(p)$ we define the set of all ports q weakly dependent from the ports p as $Weak(p) = \{q \mid p' \xrightarrow{w} q, p' \in Pre(p)\}$.

Example 36 In Figure 6.4, $Pre(a) = \{a\}$, $Pre(d) = \{a, c, d\}$ and $Pre(h) = \{e, h\}$. Moreover, $weak(a) = \{\emptyset\}$, $Weak(c) = \{b\}$ and $Weak(g) = \{b, e\}$.

Definition 22 Let a synchronous BIP component $B^f = (X, P, D)$. We define a BIP component (X, P, N) such that:

- X is the set of variables
- P is the set of ports; moreover for each port $p \in P$ the associated set of exported variables is X_p .
- the Petri net is defined as $N = (L, T, F, L_0)$, where:

– the set of places L such that

$$L = \{l_{pq,X} \mid p \xrightarrow{s} q \text{ and } X \subseteq weak(p)\} \\ \{l_{pq} \mid p \xrightarrow{w,c} q\}$$

- the set of transitions T such that

$$T = \{t_{p,X} \mid X \subseteq \text{weak}(p)\}$$
- the token flow relation $F \subset L \times T \cup T \times L$ is defined as follows:
 - * for $p \xrightarrow{s} q$, add (t_p, X, l_{pq}) , if $(q \in X)$ and (l_{pq}, t_q, Y)
 - * for $p \xrightarrow{w} q$, add (t_p, X, l_{pq}) , if $(q \in X)$ and (l_{pq}, t_q, Y) , if $(p \in Y)$
 - * for $p \xrightarrow{c} q$, add $(t_p, X, l_{pq,X})$ and $(l_{pq,X}, t_q, Y)$, if $(p \in Y)$
- L_0 is the set of initially marked places such that $L_0 \subset L$ and $L_0 = \{p \mid p \in \min P\}$.

The transformation of causal dependencies to Petri nets is represented graphically in Figure 6.7.

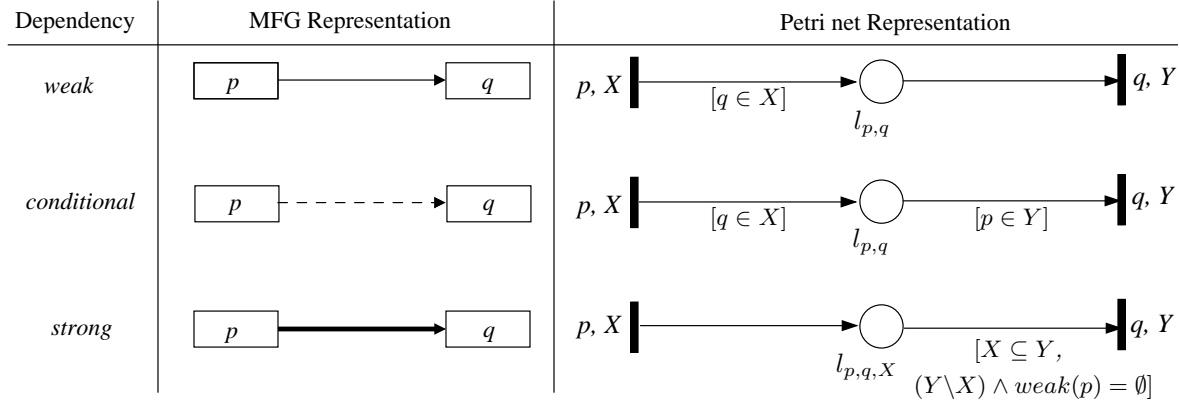


Figure 6.7: The transformation of causal dependencies to Petri nets (cluster-oriented method)

Example 37 Figure 6.8 illustrates the Petri net according to the cluster-oriented method for the example of Figure 6.4. Guards in the transitions enforce deterministic behavior restricting the execution to exactly one transition when there is the choice between two. Guards are inherited from the weak dependent ports. The produced model reproduces the form of the initial model. Clusters of ports are mapped to clusters of transitions and places and weak and conditional dependencies are represented by places. This model can be mapped to a Kahn process network where P_1, P_2 and P_3 represent processes and places l_{cb}, l_{fd}, l_{de} and l_{gh} the FIFO queues for communication between processes.

To prove the correctness of the implementation, that is, all executions of the produced Petri net conform to dependencies enforced by the modal flow graphs, we consider the following facts:

1. Each cluster of the modal flow graph is mapped to a process. A process contains those transitions that correspond to the ports of a cluster in modal flow graphs;
2. Each weak and conditional dependency is encoded to one FIFO channel. The data sent over the FIFO channel, carry the information that the source port has been executed;
3. Each process is executed involving the following steps:
 - (a) It waits to be activated. Activation of the process is done when the input FIFO channel to the initial transition (root of the cluster) has a token. This FIFO channel encodes the weak dependency that triggers the execution of the cluster. A token to this channel gives the information that the source port of the dependency has been executed and that the guard of the destination port has been evaluated to true.

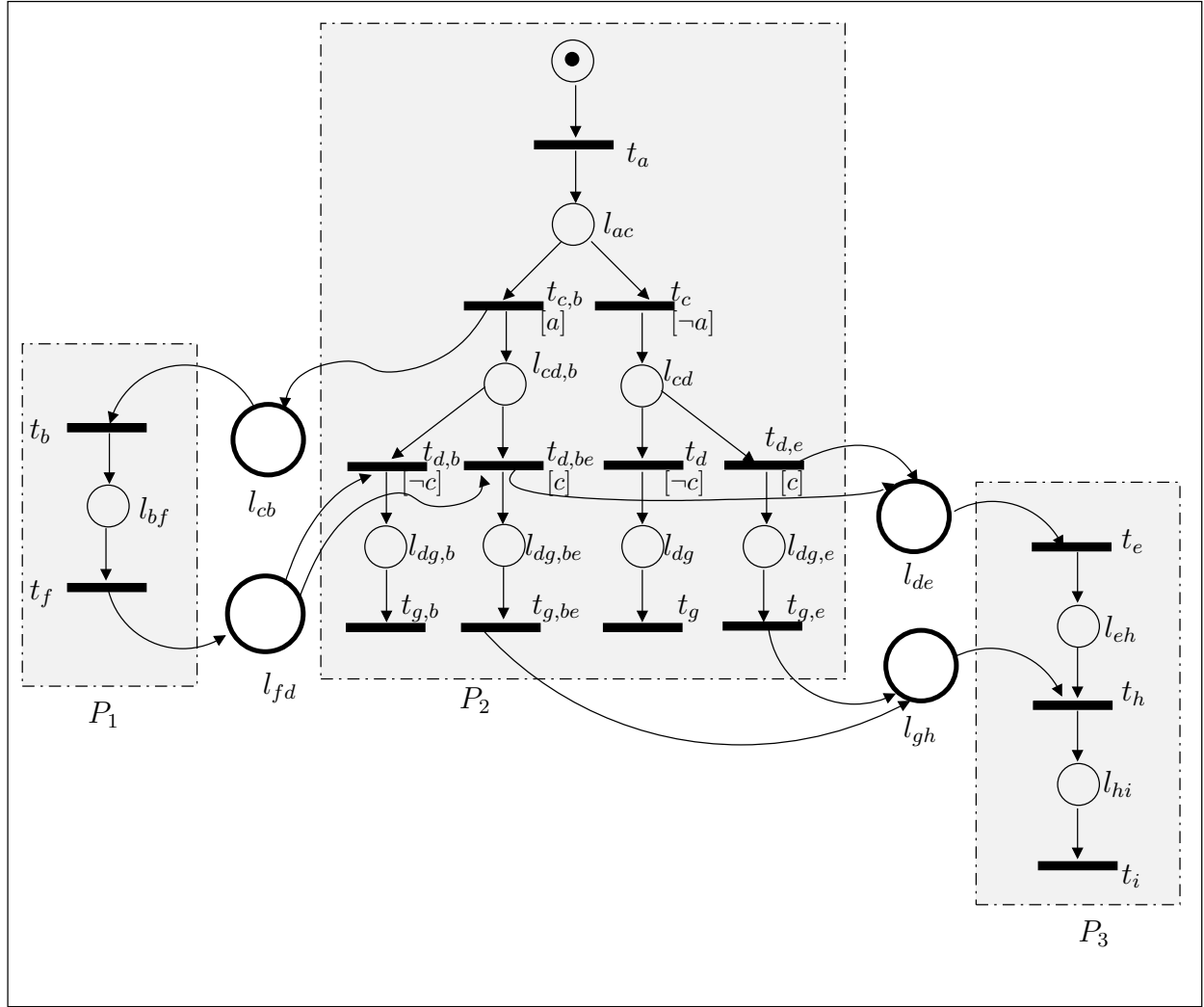


Figure 6.8: Cluster-oriented representation for the Synchronous BIP component of Figure 6.4

- (b) For each transition t_p , it reads input FIFO channels, executes the associated update function f_p and writes to output FIFO channels. Input FIFO channels correspond to conditional dependencies from other processes that have already been activated from predecessor ports of p . Output FIFO channels encode either weak dependencies to clusters that wait to be activated or conditional dependencies to processes that have already been activated from the current process.

Based on these observations, during each execution step, dependencies are implicitly examined. The causal order of dependencies is enforced and the execution of clusters triggered by ports which are not enabled is suspended. Therefore, the execution at each step produces a valid execution of the modal flow graph as defined by its semantics.

6.3 Related Work

The work in [19] is devoted to the issues of compositionality for modular code generation in dataflow synchronous languages. Causality and scheduling specification are two important features for the purpose of code generation. This work introduces the notions of endochrony and isochrony for the purposes of distributing synchronous programs on asynchronous architectures without losing semantics properties. The work in [62] extends the previous theory by introducing the notions of weak endochrony and weak isochrony. Weak endochrony allows processes within a component to run independently if no synchronization is needed. The work in [63] guarantees deterministic execution of synchronous programs in an asynchronous environment. This work defines an execution machine that generalizes the notion of weak endochrony and guarantees deterministic behavior independently of the signal absence in asynchronous environments. The solution we propose for code generation using the cluster-oriented method is close to this approach. Processes run independently and preserve their deterministic behavior.

Based on the results for endochronous systems, the work in [61] introduces a model for representing asynchronous implementations of synchronous specifications. This model covers implementations where the notion of global synchronization is preserved and in the same time globally asynchronous, locally synchronous (GALS) implementations, where global synchronization is relaxed by removing the global clock. This work provides theoretical basis that allow to reason about semantics preservation and absence of deadlock in GALS implementation of synchronous specification.

A more recent work, that is based on the endochronous design, is presented in [64]. This work introduces the clocked graph, a representation where arcs and nodes are attached with information concerning causality order and time constraints. Efficient implementation is achieved based on the Kahn principles and the notions of endochrony. The clocked graph has been used to implement distributed architectures where no global clock is considered.

The work in [30] explains how a centralized synchronous program can be executed in its environment, which is intrinsically asynchronous. For that purpose, it defines a synchronous/asynchronous interface, which links the logical time of the program with the physical time of the environment. The work [34] extends a core synchronous data-flow language with a notion of periodic clocks, and designs a relaxed clock calculus (a type system for clocks) to allow non strictly synchronous processes to be composed or correlated. The work [54] presents methods to generate modular code from synchronous block diagrams. That is, for a given block of the diagram, code is generated independently from context. Modular code allows reusability of blocks without creating dependency cycles.

6.4 Discussion

The previous sections described two methods for distributed implementation of Synchronous BIP models. Both methods consist of two steps. First, transformation of modal flow graphs to particular classes of Petri nets. Both transformations ensure deterministic behavior and enforce causality for dependencies between ports. 2) Mapping of produced Petri nets to Kahn process networks.

The “direct” method uses a global synchronization, represented by the t_{sync} transition. The execution of t_{sync} denotes the end of a step and the beginning of a new one. That is, at each step, the computation of the model is completed by executing the process that corresponds to the t_{sync} transition. This action can be seen as a resynchronization of the model.

Another aspect of the direct method is that there is a continuous exchange of data between communicated processes. FIFO channels send data containing information about the execution status of the ports. The network is obliged to keep the communication between processes even if only negative transitions are executed.

The “cluster-oriented” method preserves the initial clusterized design. This design decomposes the model to clusters of ports. Clusters wait to be activated from other clusters. This design shows clearly the different execution times within a model. Communication between clusters is done using FIFO channels. Empty input FIFO channels for the initial transitions of clusters suspend the whole execution of the cluster till a token arrives.

Chapter 7

Representation of Latency-Insensitive Designs in Synchronous BIP

The theory of latency-insensitive design (LID) proposed by Carloni et al. [29] deals with the problem of communication latencies between synchronous components.

Synchronous systems assume that computation and communication takes "no time". However, in hardware design the situation is not exactly the same. The latest trends want scaled chips to dominate in the world of digital systems. Wire delay increases with scaled chips, that means that signals will need more than one clock cycle to be propagated along wires. Latency of long wires can be critical for complex systems.

According to the LID theory, synchronous systems can be "desynchronized" as networks of synchronous processes that might run with increased frequency. Specific interconnect mechanisms are introduced to "resynchronize" the global system. These methodologies, called *relay stations* and *shell wrappers*, allow latencies at interconnects between processes.

The theory of latency-insensitive design has been formalized using the tagged-signal model [53]. This model provides denotational semantics for describing the computation of synchronous and asynchronous systems.

In this chapter we present an alternative method for representation of latency-insensitive designs as Synchronous BIP systems. Section 7.1 describes the methodology of the Latency-Insensitive Design as proposed in [29]. The single-clock Synchronous Design is presented in section 7.2. These are Synchronous BIP components restricted to a single activation port. For the sequel, we consider synchronous systems consisting of exclusively single-clock components. Section 7.3 presents the transformation of the Latency-Insensitive Design to Synchronous BIP components. Conclusions are drawn in section 7.4.

7.1 The Methodology

This section presents the basic aspects of the latency-insensitive design (LID). We use the following notations:

- An *event* is a member of $\mathcal{V} \times \mathcal{T}$, where \mathcal{V} is a set of values and \mathcal{T} is a set of timestamps. A *signal* is a set of events. A subset of N-tuples of signals is a *process*.

- An absent event (\perp) is called *stalling* event. An event which is not *stalling* is called *informative* event.
- A process is called *strict* if all informative events precede all *stalling* events.
- A process is called *stallable* if stalling events are inserted synchronously on all input signals and all output signals.
- A process which tolerates arbitrary distribution of stalling events (delays) among its signal is called *patient*.

The LID approach proposes a solution, which does not require a costly redesigning of the system nor adaptation to the latencies of the systems. An initial model consists of strict synchronous processes. The LID design aims to transform every *strict* process to a *patient* one by adding a set of auxiliary modules. The procedure is done in two steps:

1. It breaks long delay interconnections into segments such that each of the segments can transmit signals in one clock cycle. Each segment is implemented by a buffer of capacity at least two, called *relay station*.
2. It encapsulates synchronous processes by a set of modules called *shell wrapper* to ensure the correct synchronization of the in/out data-flow. Each strict process receives and sends only valid data (informative events). In order for the system to achieve the expected behavior, processes must be stallable, that is, all inputs and all outputs have the same delays.

Relay Stations

Relay stations are channels which act as media of communication between the synchronous processes. They are implemented with buffers in order to store data. A buffer of size two is the minimum capacity buffer for achieving maximal throughput. At each cycle, the stored data is transferred and the current read value is stored, avoiding overwriting or loss of data.

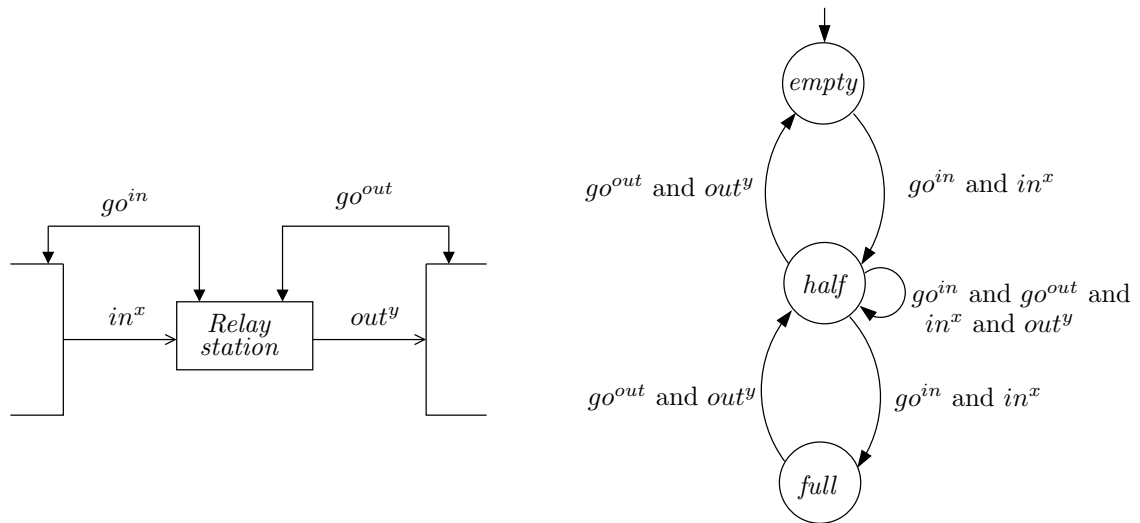


Figure 7.1: Relay Station Structure and the corresponding FSM

Figure 7.1 shows the interface of a relay station. An input data signal in^x is read whenever the control signal go^{in} is received. Similarly, an output signal out^y is produced if the control signal go^{out} is received.

The FSM of Figure 7.1 shows the behavior of a relay station. Initially, the buffer of a relay station is empty. That is, the relay station can only read input data. At the next step and considering that its buffer has a stored value, the relay station can produce this data and/or read a new one. When the buffer is full, it can only produce an output data. In general, a go^{in} signal is emitted if the buffer has at least one free place to store data, that is, either it is in state empty or half. A go^{out} signal is emitted if the buffer has at least one data stored, that is, either in state half or full. The emission of a control signal (go^{in}/go^{out}) is accompanied by the emission of the corresponding data signal (in^x/out^y), that is the transfer (in/out) of data.

Wrappers

A set of functional processes are composed together with a synchronous process (strict and stallable) P in order to produce a *patient* process W , called *shell wrapper*. The shell wrapper W and the process P produce the same sequence of output data for the same input data independently of the delays. The shell wrapper guarantees that outputs are not produced unless all inputs are valid (informative events).

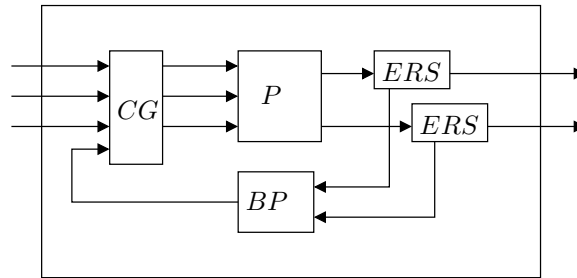


Figure 7.2: Wrapper Structure - P is a *stallable* synchronous process which reads three inputs and produces two outputs. It is encapsulated by a clock gate (CG), a back pressure (BP) and two extended relay stations (ERS) one for each output of the process P

Each *shell wrapper*, as shown in Figure 7.2, consists of the following three modules:

- a *clock gate*, it guarantees that the process P will be activated only when all inputs are received. That is, it aligns input flows and generates the activation events that trigger the execution of the process
- one *extended relay station* for each output of the synchronous process. It is a two places buffer strongly synchronized with the process P . It stores output values produced by the synchronous process.
- a *back pressure*, it synchronizes the relay stations with the clock gate such that the process P reads inputs only when the outputs can be consumed, in the same clock cycle, by the extended relay stations.

7.2 The Single-clock Synchronous Design

A Synchronous BIP component is called *single-clock* if there is a single activation port that triggers the execution of the component. Moreover the dependencies between ports are restricted to strong. These components corresponds to LUSTRE nodes, considering only single-clock operators (see Chapter 4). Figure 7.3 shows an example of a *single-clock* Synchronous BIP component.

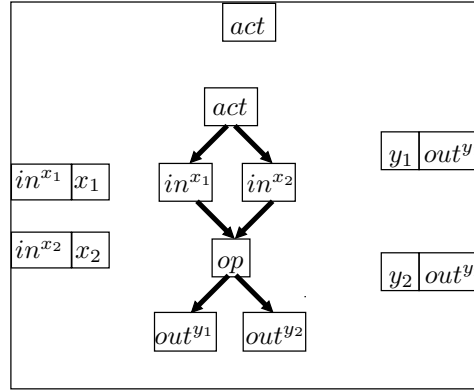


Figure 7.3: A *single-clock* Synchronous BIP component. At each step, the component is triggered by the port act , reads the two inputs at the ports in^{x1} and in^{x2} , performs an operation op and writes the two output at ports out^{x1} and out^{x2} .

The following definition describes formally a *single-clock synchronous component*.

Definition 23 (Single-clock Synchronous Component) A *single-clock synchronous component*, noted B^s satisfies the additional properties:

1. The component has a unique control port act which is the root of the component;
2. The control port act triggers the execution of all data ports $\{in^{x1}, \dots, in^{xi}, out^{y1}, \dots, out^{yj}\}$;
3. All dependencies are strong.

A single-clock synchronous component satisfies the properties of unique strong cause (property (2)) and exclusively either strong or weak causes for each port (property (3)), thus it is *well-triggered*.

A single-clock synchronous system is realized by the composition of single-clock synchronous components. Composition of single-clock synchronous components is characterized by a strong synchronization γ^{act} involving control ports act of all components. Moreover, data ports are synchronized through data flow interactions that connect one out^y port to one in^x port. The formal definition of a single-clock synchronous system is given below.

Definition 24 (Single-clock Synchronous System) A *single-clock synchronous system* is the composition of single-clock synchronous components with the following two types of interactions:

- control ports of a synchronous system are strongly synchronized to a unique interaction γ^{act} ,
- data ports of different strict synchronous systems are communicating through data flow interactions of type $\gamma^{io} = \{in^x, out^y\}$

7.3 Transformation of Synchronous BIP Systems to LID

This section provides the transformation of single-clock synchronous systems into LID systems. We provide models for *relay stations* and *shell wrappers* as synchronous BIP components.

Breaking data flow connectors

The synchronous component that corresponds to a relay station (Figure 7.4) contains a two places buffer B . The component exchanges activation events with its environment through the ports $\{go^{in}$ and $go^{out}\}$. The go^{in} port is executed if the buffer is not full and triggers the execution of the data port in^x (strong dependency). The in^x reads a value from its environment and adds it to the buffer B . The go^{out} port is executed if the buffer is not empty. It triggers the execution of the port out^y (strong dependency). This port propagates one value of the buffer at its environment. The control for the current status of the buffer is done before reading new values or propagating already existing ones. If both activation ports can be triggered in one step, then they are enforced to be executed before any data port is executed (conditional dependencies).

Definition 25 (Relay Station) A relay station synchronous component B^{rs} is defined by the tuple (X, P, D) where

- $X = \{B, x, y\}$ is the set of data variables with B a FIFO buffer of two places and x, y data exported through the ports in^x and out^y .
- P is the set of control ports $\{go^{in}, go^{out}\}$ and of data ports $\{in^x, out^y\}$
- D is the set of dependencies as shown in figure 7.4

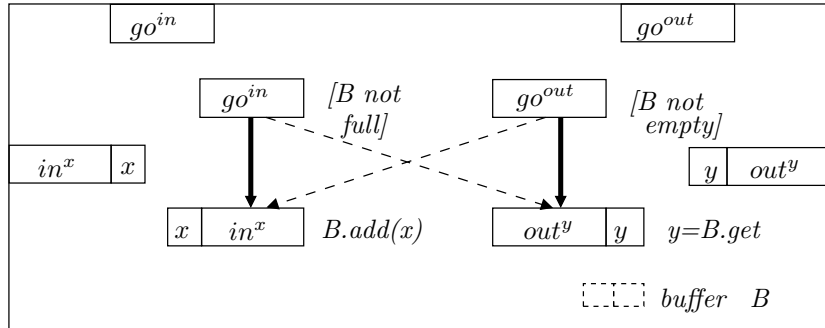


Figure 7.4: A relay station represented by a synchronous BIP component

Relay stations are used to “break” in/out connectors.

Figure 7.5 illustrates the functionality of a relay station. On the top of the Figure, processes A and B are strongly synchronized through the control ports act and through a data connector between the ports in and out respectively. On the bottom of the Figure, the data connector is replaced by a sequence of interconnected relay stations $B_1^{RS}, B_2^{RS}, \dots, B_k^{RS}$ where $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)$ correspond to the buffers B_1, B_2, \dots, B_k respectively.

Initially, all buffers are empty. At the first clock cycle, the component A produces z_1 . This value is stored at the relay station B_1^{RS} . At the second clock cycle, the component A produces a new value z_2 . The value z_1 is then propagated at the relay station B_2^{RS} and the value z_2 is stored

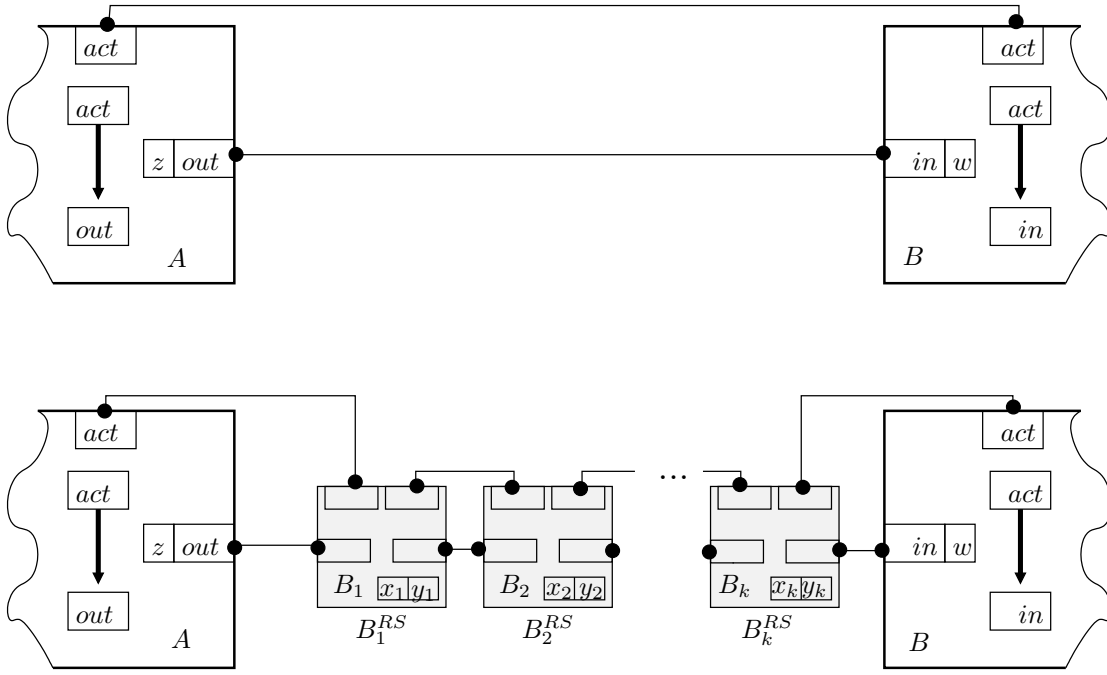


Figure 7.5: Breaking an in/out connector to a sequence of relay stations

at the relay station B_1^{RS} . At the k th cycle, the value z_1 will be propagated at the k th buffer. Finally, at the k th+1 cycle, the value z_1 , if requested, can be consumed by the component B.

An instant of the execution is shown in Figure 7.6. It can be shown that the decomposition of the in/out connector to relay stations is equivalent with the initial connector. That is, the data sent from the *out* port of the component and put on the buffers of the relay stations are eventually read from the component B after a number of steps and in the right order (the order in which they were sent).

clock cycle	0	1	2	...	k	k+1	k+2	...
z	\perp	z_1	z_2	...	z_k	z_{k+1}	z_{k+2}	...
$B_1(x_1, y_1)$	(\perp, \perp)	(\perp, z_1)	(\perp, z_2)	...	(\perp, z_k)	(\perp, z_{k+1})	(\perp, z_{k+2})	...
$B_2(x_2, y_2)$	(\perp, \perp)	(\perp, \perp)	(\perp, z_1)	...	(\perp, z_{k-1})	(\perp, z_k)	(\perp, z_{k+1})	...
...
$B_k(x_k, y_k)$	(\perp, \perp)	(\perp, \perp)	(\perp, \perp)	...	(\perp, z_1)	(\perp, z_2)	(\perp, z_3)	...
w	\perp	\perp	\perp	...	\perp	z_1	z_2	...

Figure 7.6: Instants of the execution for the design of Figure 7.5

Wrapping of components

The role of the *wrapper* is to encapsulate a synchronous process into a *patient* process, that is, a functional process insensitive to the delays of the inputs. It consists of three modules, the *clock gate*, the *back pressure* and the *extended relay stations*.

In the sequel, we give some formal definition for these modules described in Synchronous BIP.

Definition 26 (Clock gate) A clock gate *synchronous BIP component*, denoted as B^{cg} , is

defined by the tuple (X, P, D) where:

- $X = \{b_1, \dots, b_n, x_1, \dots, x_n\}$ is the set of Boolean variables b_1, \dots, b_n with initial values false and the variables x_1, \dots, x_n (one for each input/output).
- P is the set of ports, $\{go, act, go^{in_1}, \dots, go^{in_n}, in^{x_1}, \dots, in^{x_n}, out^{x_1}, \dots, out^{x_n}\}$. Through the $go^{in_1}, \dots, go^{in_n}$ control ports the wrapper gets synchronized with relay stations. Through the $in^{x_1}, \dots, in^{x_i}$ ports the wrapper receives inputs from relay stations. A ready event is received from the back pressure. An act signal is the clock of the synchronous process and produced if all input data and the ready event are received. Finally, the data ports $out^{x_1}, \dots, out^{x_n}$ produce the output values that were received via the input ports $in^{x_1}, \dots, in^{x_n}$.
- D is the set of dependencies as shown in figure 7.7.

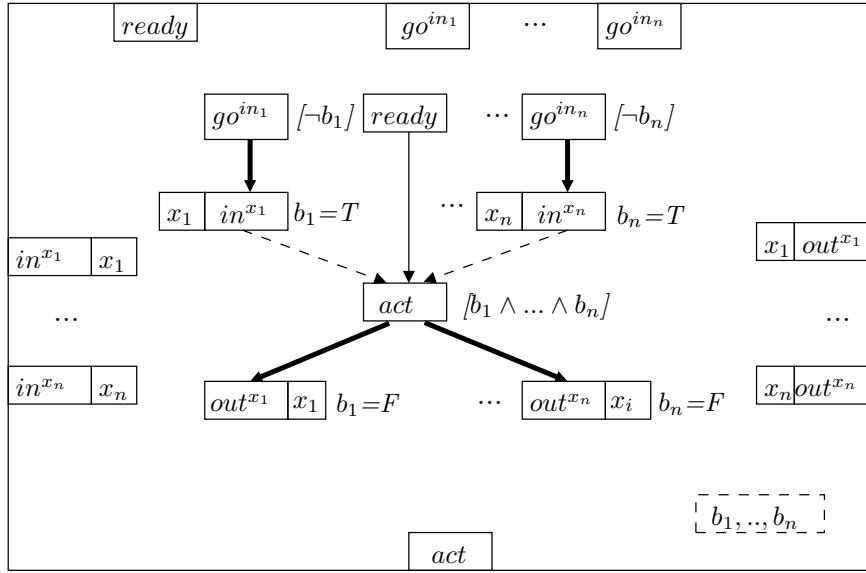


Figure 7.7: A clock gate represented by a synchronous component

The *clock gate* aligns a set of inputs and provides them to the synchronous process. The strong dependencies between data ports go^{in_i} and control ports in^{x_i} , say that new data may be received if the variables b_i are set to false. When an inputs x_i is read through a port in^{x_i} , the corresponding variable b_i becomes true. When all inputs are read and the control port $ready$ is enabled, the control port act is triggered and the outputs x_i are sent through the ports out^{x_i} . Then, all variables b_i are set to false and a new synchronous step is ready to start.

Definition 27 (Extended relay station) An extended relay station in synchronous BIP is a component denoted as B^{ers} and it is defined by the tuple (X, P, D) where:

- the set of data variables X is a FIFO buffer of capacity two
- P is the set of control ports $\{go^{in}, go^{out}, act\}$ and of data ports $\{in^x, out^y\}$. The control port go^{out} and the data port in^x synchronize the wrapper with a relation station which will receive an output produced by the wrapper.
- D is the set of dependencies as shown in figure 7.8.

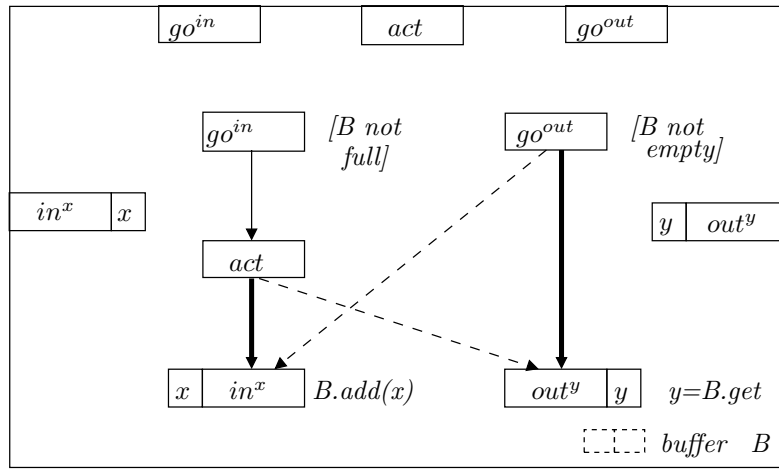


Figure 7.8: An extended relay station represented by a synchronous component

The behavior of an *extended relay station* is similar to this of a relay station. Moreover, an *extended relay station* has an additional control port, named *act* for strong synchronization with the synchronous process.

Definition 28 (Back pressure) A back pressure *synchronous component* B^{bp} is defined by the tuple (P, D) where the port *go* is weakly dependent on the set of ports $\{ready^{in_i}\}_{i=1,k}$ as shown in figure 7.9.

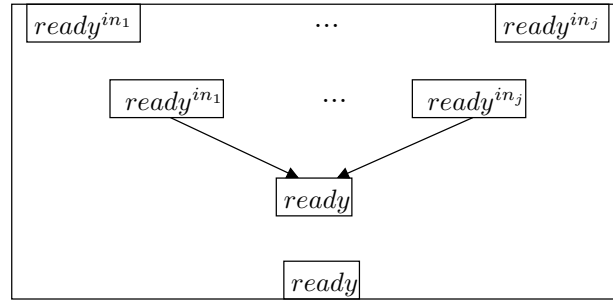


Figure 7.9: A back pressure Synchronous BIP component

This component produces a unique activation event *ready* if all events $ready^{in_1}, \dots, ready^{in_j}$ are received. These are events sent by the *extended relay stations* to denote their availability for storing data. The activation event *ready* of the *back pressure* is strongly synchronized with the *clock gate*. When the ready event is generated, the clock gate reads input data.

Figure 7.10 shows a shell wrapper for the single-clock component *pre* of Figure 7.3.

The cyclic BIP component of Figure 7.3 is encapsulated by the components clock gate B^{cg} , extended relay stations B^{ers_1} and B^{ers_2} and back pressure B^{bp} . Atomic components are strongly synchronized with connections as shown in the Figure. Composing atomic components, we obtain a unique component as shown in Figure 7.11.

The Figure depicts the different modules of the shell wrapper. On the top, back pressure and clock gate, in the middle the single-clock process and white ports correspond to one extended relay station. We can remark that inputs and outputs are “desynchronized” since they take

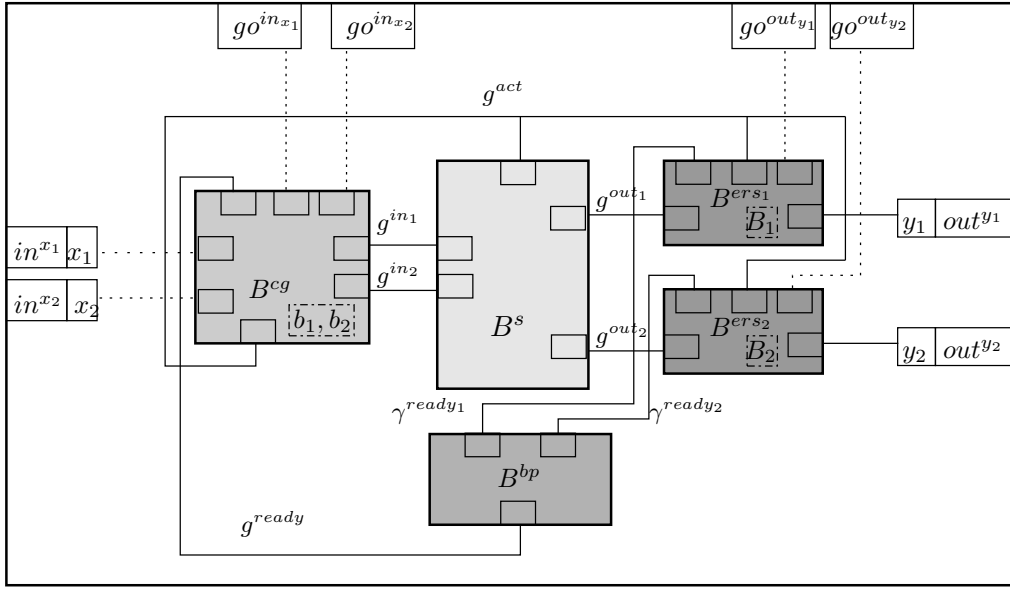


Figure 7.10: Shell wrapper compound component for the single-clock component of Figure 7.3

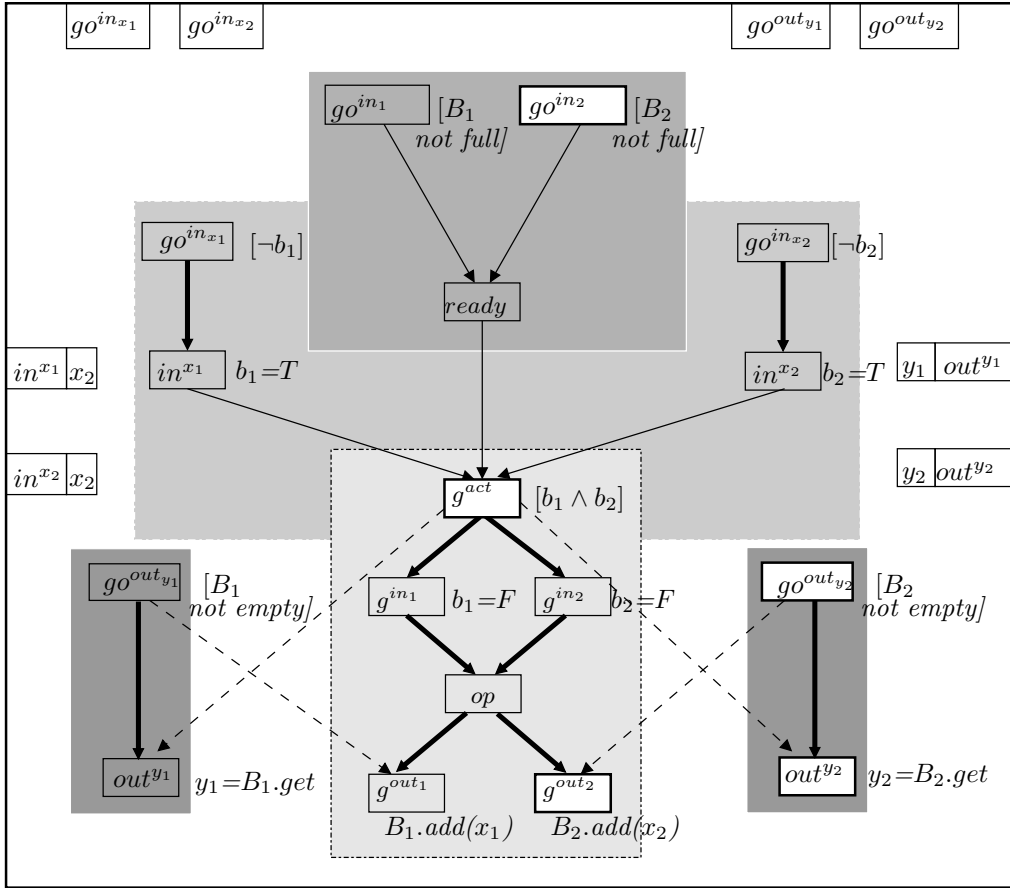


Figure 7.11: Composition of shell wrapper of Figure 7.10

place on different activation ports (go^{in_x} and go^{out_y} respectively). The function (op) is activated when all inputs are present ($[b_1 \wedge b_2]$) and when there is space in all out buffers ($[B_1 \text{ not full}]$ and $[B_2 \text{ not full}]$).

Comparing the composed component with the initial single-clock component (see Figure 7.3), we observe that the order of execution of ports remains the same. Thus, it can be shown that the LID constructions of Synchronous BIP components produces equivalent models.

7.4 Discussion

This chapter described a representation of latency-insensitive designs (LID) in Synchronous BIP. According to the LID theory, synchronous systems can be “desynchronized” as networks of synchronous processes that might run with increased frequency. *Relay stations* and *Shell Wrappers* are modules introduced by the LID theory in order to “resynchronize” the global system. We show how to map those modules in Synchronous BIP components and the functional equivalence between the initial and the transformed model.

Chapter 8

Conclusion

Achievements

We have presented a general approach for modeling synchronous component-based systems. These are systems of synchronous components strongly synchronized by a common action *sync* that initiates execution steps of each component. Steps are described by priority Petri nets. Priorities are instrumental for enforcing run-to-completion in the execution of a step. *Modal flow graphs* have been introduced and used to define a particular class of Petri nets for which deadlock-freedom and confluence are met by construction provided some easy-to-check conditions hold. This result is the generalization of existing results for classes of Petri nets without conflicts. It allows more general behavior for components given that the semantics of conditional dependencies lead to Petri nets with backward conflicts and priorities.

We have applied the construction of Synchronous BIP components to the LUSTRE synchronous language. The result is a semantic preserving mapping of LUSTRE into BIP. This mapping shows the interplay between data flow and control flow and allows understanding how strict synchrony can be weakened to get less synchronous computation models. We have also shown a translation from the discrete-time fragment of Simulink into Synchronous BIP. The translation is structural and incremental. The Synchronous BIP components obtained by the translation of Simulink models have several properties including confluence and deadlock-freedom.

We have provided the Synchronous BIP toolset, an extension of the BIP toolset. Figure 8.1 illustrates an overview of the Synchronous BIP toolset. This toolset includes the Synchronous BIP language, Language Factories, Synchronous BIP compiler and code generators. The *Synchronous BIP language (S-BIP)* provides constructs for describing synchronous systems in the Synchronous BIP framework. It reuses constructs of the BIP language and introduces some new ones to describe the behavior of modal flow graphs. The *Language Factories* contain transformations from synchronous formalisms into Synchronous BIP. Currently, the Language Factories for Synchronous BIP contains two tools, the *Lustre2-SBIP* for translating LUSTRE programs to Synchronous BIP and the *Simulink2-SBIP* for the translation of Simulink models into Synchronous BIP.

We have presented how to generate sequential and distributed code from Synchronous BIP models. The sequential implementation produces endless single loop C code. The results obtained by measuring performances on LUSTRE and Simulink examples are comparable with the results produced by code generators of LUSTRE and Simulink respectively. Moreover, we have observed clear overhead of the endless loop C code produced by the Synchronous BIP compiler and the C code produced by the BIP compiler. For the distributed implementation, we have

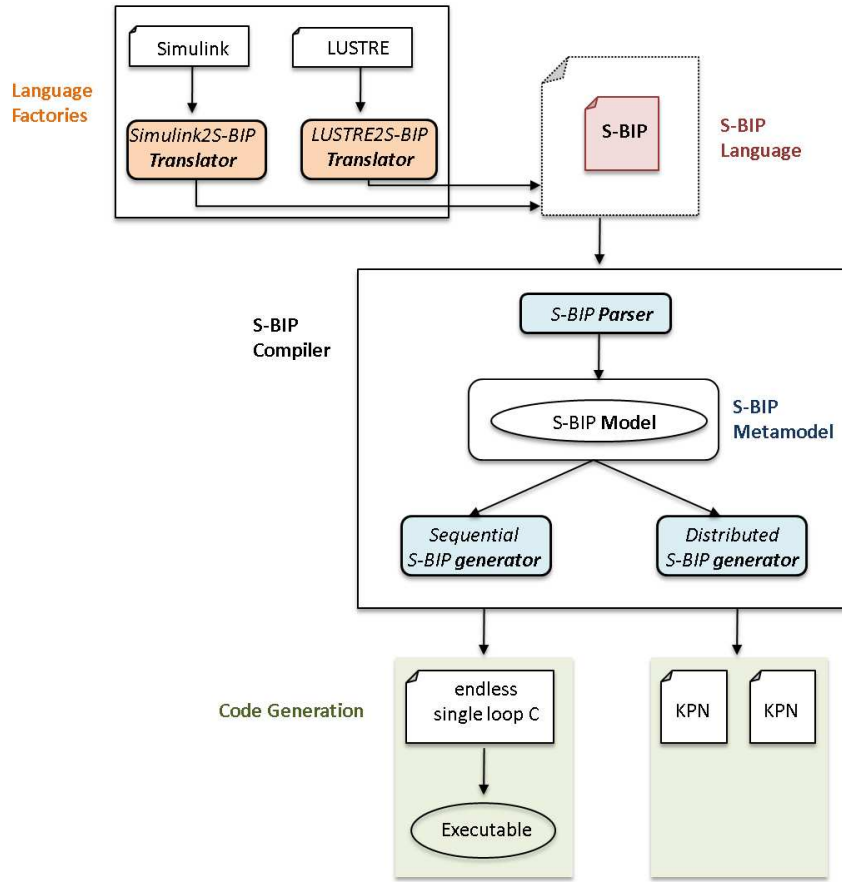


Figure 8.1: The Synchronous BIP toolset

presented two methods, the *direct* method and the *cluster-oriented* method. Both methods proceed in two steps. First, they transform modal flow graphs to particular classes of Petri nets. Second, they map these Petri nets to Kahn process networks.

Finally, we have proposed a latency-insensitive design for Synchronous BIP. We have been based on the work done by Carloni et.al and we proposed representation of *relay stations* and *shell wrappers* as Synchronous BIP components.

Perspectives for Future Work

The main goal of this thesis was only partially achieved. The Synchronous BIP framework we introduced in this work, allows integration of synchronous systems theory in an all encompassing component framework without losing advantages such as correctness-by-construction and efficient code generation. This makes possible modeling mixed synchronous/asynchronous systems without artefacts. The definition of synchronous components as a subset of the BIP framework allows their combination with other asynchronous languages that can be translated into BIP. However, because of lack of case studies, we were not able to demonstrate a possible integration of synchronous and asynchronous systems. It remains a very interesting problem to be studied in the future.

The translation principles for LUSTRE and Simulink can be generalized for other synchronous formalisms. We plan to propose more translations of that kind creating a library of components from synchronous formalisms into Synchronous BIP.

As far as the translation from Simulink to Synchronous BIP concerns, although we covered

a significant part of the discrete-time fragment of Simulink, our translation is not complete and can be rapidly extended in several directions. On a longer term perspective, we would like to extend the translation from Simulink to Synchronous BIP to the full discrete-time fragment. This must include all of the conditionally executed subsystems, like the triggered and enabled subsystems, the function-call subsystems as well as user defined functions blocks. We plan to define a similar translation for discrete-time Stateflow. Finally, we plan to extend the translation for the continuous-time fragment of Simulink. This last direction needs extension of the BIP model to encompass for continuous computation.

The methods we proposed for distributed code generation are in preliminary stages. We plan to continue the work on that domain by providing full implementations and experimental results. Finally, further research is required for the latency-insensitive design of BIP. Formal validation and correctness of the transformation will be provided in future work.

Bibliography

- [1] <http://www.laas.fr>.
- [2] <http://www.mathworks.com/products/simulink/>.
- [3] <http://www-verimag.imag.fr/ss2lus.html/>.
- [4] <http://www-rocq.inria.fr/scicos//>.
- [5] <http://groups.csail.mit.edu/cag/streamit/index.shtml>.
- [6] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 229–238, New York, NY, USA, 2010. ACM.
- [7] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques*, page 2004, 2004.
- [8] C. André. Synccharts: a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.
- [9] A. Basu. *Component-based Modeling of Heterogeneous Real-time Systems in BIP*. PhD thesis, UJF, 2008.
- [10] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, FORTE '08, pages 116–133, Berlin, Heidelberg, 2008. Springer-Verlag.
- [11] A. Basu, M. Bozga, S. Bensalem, B. Caillaud, B. Delahaye, and A. Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *Proc 30th International Conference on Formal Techniques for Distributed Systems, Amsterdam*, volume 6117 of *Lecture Notes in Computer Science*, pages 32–46. Springer-verlag, 2010.
- [12] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
- [13] A. Basu, M. Gallien, C. Lesire, T. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis. Incremental Component-Based Construction and Verification of a Robotic System. In *European Conference on Artificial Intelligence ECAI'08 Proceedings*, volume 178 of *FAIA*, pages 631–635. IOS Press, 2008.

- [14] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using BIP for Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks. In *Proceedings of NCA '07*, pages 257–260, 2007.
- [15] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09*, pages 614–619, Berlin, Heidelberg, 2009. Springer-Verlag.
- [16] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991.
- [17] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM Trans. Embed. Comput. Syst.*, 7:43:1–43:36, August 2008.
- [18] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.
- [19] A. Benveniste, P. L. Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: specification & distributed code generation, 1997.
- [20] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16:103–149, September 1991.
- [21] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to estereel. In *Proceedings of the first Franco-Japanese Symposium on Programming of future generation computers*, pages 35–56, Amsterdam, The Netherlands, The Netherlands, 1988. Elsevier Science Publishers B. V.
- [22] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. Compilation of Polychronous Data Flow Equations. In Sandeep K. Shukla and Jean-Pierre Talpin, editors, *Synthesis of Embedded Software*, pages 1–40. Springer, 2010.
- [23] S. Bliudze and J. Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.
- [24] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software, EMSOFT '10*, pages 209–218, New York, NY, USA, 2010. ACM.
- [25] M. Bozga. *Component-based Design of Real-time Systems*. PhD thesis, UJF, 2009.
- [26] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in bip.
- [27] M. Bozga and E. Sifakis. Issues on memory-management for component-based systems. In *Exploiting Concurrency Efficiently and Correctly, EC²*, 2010.
- [28] M. D. Bozga, V. Sfyrla, and J. Sifakis. Modeling synchronous systems in bip. In *Proceedings of the seventh ACM international conference on Embedded software, EMSOFT '09*, pages 77–86, New York, NY, USA, 2009. ACM.

- [29] L. P. Carloni, S. Member, K. L. Mcmillan, and A. L. Sangiovanni-vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:1059–1076, 2001.
- [30] P. Caspi and A. Girault. Execution of distributed reactive systems. In *Proceedings of the First International Euro-Par Conference on Parallel Processing*, Euro-Par '95, pages 15–26, London, UK, 1995. Springer-Verlag.
- [31] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *Software Engineering, IEEE Transactions on*, 25(3):416–427, 1999.
- [32] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Models in software engineering. chapter Translating AADL into BIP - Application to the Verification of Real-Time Systems, pages 5–19. Springer-Verlag, Berlin, Heidelberg, 2009.
- [33] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. *SIGPLAN Not.*, 41:180–193, January 2006.
- [34] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. *SIGPLAN Not.*, 41:180–193, January 2006.
- [35] P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. In *In Proc. ATVA 2005, number 3707 in LNCS*, pages 414–428. Springer, 2005.
- [36] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. Syst. Sci.*, 5:511–523, October 1971.
- [37] J. Eidson, M. Fischer, and J. White. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *34 th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 243–254, 2002.
- [38] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.
- [39] A. Gamatié, T. Gautier, P. L. Guernic, and J.-P. Talpin. Polychronous design of embedded real-time applications. *ACM Trans. Softw. Eng. Methodol.*, 16(2), 2007.
- [40] G. Gössler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55:161–183, March 2005.
- [41] N. Halbwachs. About synchronous programming and abstract interpretation. *SCP*, 31(1):75–89, 1998.
- [42] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.
- [43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of IEEE*, 79(9):1305–1320, 1991.

- [44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.
- [45] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.
- [46] R. Harper and C. Stone. A type-theoretic interpretation of standard ml. In *In Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.
- [47] M. W. Heath, W. P. Burleson, and I. G. Harris. Synchro-tokens: A deterministic gals methodology for chip-level debug and test. *IEEE Transactions on Computers*, 54:1532–1546, 2005.
- [48] M. Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP*. PhD thesis, UJF, 2010.
- [49] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.
- [50] K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *Proceedings of the 16th European conference on Programming*, ESOP’07, pages 64–79, Berlin, Heidelberg, 2007. Springer-Verlag.
- [51] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.
- [52] E. A. Lee and D. G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON*, pages 310–315, 1987.
- [53] E. A. Lee and A. Sangiovanni-vincentelli. The tagged signal model - a preliminary version of a denotational framework for comparing models of computation. Technical report, University of California, Berkeley, CA, 1996.
- [54] R. Lublinerman and S. Tripakis. Modularity vs. reusability: code generation from synchronous block diagrams. In *Proceedings of the conference on Design, automation and test in Europe*, DATE ’08, pages 1504–1509, New York, NY, USA, 2008. ACM.
- [55] F. Maraninchi and T. Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE ’07, pages 53–62, New York, NY, USA, 2007. ACM.
- [56] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.
- [57] N. Marian and S. Top. Integration of simulink models with component-based software models. *Advances in Electrical and Computer Engineering*, 2008.
- [58] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating simulink models into input language of a model checker. In *ICFEM*, pages 606–620, 2006.
- [59] D. Nowak. Synchronous structures, 1999.

- [60] S. B. K. H. J. S. P. Bourgos, A. Basu. Integrating architectural constraints in application software by source-to-source transformation in bip. Technical Report TR-2011-1, Verimag Research Report, 2010.
- [61] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundam. Inf.*, 78:131–159, January 2007.
- [62] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Form. Methods Syst. Des.*, 28:111–130, March 2006.
- [63] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 124–133, New York, NY, USA, 2007. ACM.
- [64] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 147–156, New York, NY, USA, 2009. ACM.
- [65] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268, New York, NY, USA, 2004. ACM.
- [66] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of simulink models into synchronous bip. In *SIES*, pages 217–220, 2010.
- [67] J. Sifakis. A framework for component-based construction extended abstract. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 293–300, Washington, DC, USA, 2005. IEEE Computer Society.
- [68] G. Smeding and G. Goessler. Modal flow graphs to petri nets. Personal Communication.
- [69] C. Sofronis. *Embedded Code Generation from High-Level Heterogeneous Components*. PhD thesis, UJF, 2006.
- [70] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, 2005.