Modeling self-configuration in Architecture-based self-adaptive systems

Rim El Ballouli

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Rim El Ballouli

Thèse dirigée par **Saddek Bensalem**, Université Grenoble Alpes

préparée au sein du **Laboratoire Verimag**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique (MSTII)**

Modeling Self-configuration in Architecture-based Self-adaptive systems

Modélisation de la configuration automatique dans des systèmes auto-adaptifs basés sur l'architecture

Thèse soutenue publiquement le **20 Mars 2019**,
devant le jury composé de :

M. Kamel Barkaoui
Professeur, CNAM Paris, Président du jury et Rapporteur
M. Iulian Ober
Maître de conférence, Université Toulouse Jean Jaurès, Rapporteur
M. Simon Bliudze
Chargé de recherche, INRIA Délégation Lille Nord Europe, Examinateur
M. Martin Wirsing
Professor, Université Louis et Maximilians de Munich, Examinateur
M. Markus Roggenbach
Professor, Université de Swansea au pays de Galles, Examinateur
M. Marius Bozga
Ingénieur de recherche, CNRS Délégation Alpes, Examinateur
M. Saddek Bensalem
Professeur, Université Grenoble Alpes, Directeur de thèse

Acknowledgements

First and foremost, I would like to express my deep gratitude to Professor Saddek Bensalem whom under his supervision I was given an opportunity to explore a completely new domain to me. I thank him for highlighting valuable research direction and providing a healthy working environment that helped me achieve this work.

I am grateful to Dr. Marius Bozga for giving me just enough space to explore things on my own and stepping in for guidance whenever needed. I thank him for his availability, constructive criticism and foremost his patience in answering my repeated questions. A special thanks to Professor Joseph Sifakis, for his valuable input and for always pushing to bring out only the best of me.

Thanks to colleges at Verimag and friends for their support. A special thanks to Mona Darwish for being there every step of the way, Zeina Habli for always bringing me back to my senses and Lina Aliouat who was more excited than I was for finalizing the dissertation.

Last but not least, I thank my family for their unconditional support, scarifies, and their continuous encouragement that kept me driving forward despite the friction.

To my parents,
who whispered in my ears

"knowledge is thy armor"

Abstract

Modern systems are pressured to adapt in response to their constantly changing environment to remain useful. Traditionally, this adaptation has been handled at down times of the system. there is an increased demand to automate this process and achieve it whilst the system is running. *Self-adaptive systems* were introduced as a realization of continuously adapting systems. Self-adaptive systems are able to modify at runtime their behavior and/or structure in response to their perception of the environment, the system itself, and their requirements. The focus of this work is on realizing self-configuration, a key and essential property of self-adaptive systems. Self-configuration is the capability of reconfiguring automatically and dynamically in response to changes. This may include installing, integrating, removing and composing/decomposing system elements.

This thesis introduces the Dr-BIP framework, an extension of the BIP framework for modeling self-configuring systems that relies on a model-based and component & connector approach to prescribe systems. The combination of both of these approaches exploits the benefits of each.

A Dr-BIP system model is a runtime model which captures the running system at three different levels of abstraction namely behavior, configuration, and configuration variants. The system's configuration is captured by component and connectors. In a component and connector system, self-configuration can have three different levels of granularity which includes the ability to add or remove connectors, add or remove components, and add or remove subsystems. Dr-BIP supports explicit addition and removal of both components and subsystems, but implicit addition and removal of connectors. The main advantage of relying on an implicit addition and removal of connectors is the ability to guarantee by construction specific configuration topologies.

To capture the three levels of abstraction, we introduce motifs as primary structures to prescribe a self-configuring Dr-BIP system. A motif defines a set of components that evolve according to interaction and reconfiguration rules. A system is composed of multiple motifs that possibly share components and evolve together. Interaction rules dictate how components composing the system can interact and reconfiguration rules dictate how the system configuration can evolve over time. Finally, we show that the proposed framework is both minimal and expressive by modeling four different self-configuring systems. Last but not least, we propose a modeling language to codify the framework concepts and provision an interpreter implementation.

Keywords: *model driven engineering, reconfigurable, dynamic, self-configuring, self-adaptive, component and connector architectures*

Résumé

Pour rester utile, les systèmes modernes doivent s'adapter à leur environnement qui ne cessent d'évoluer. Traditionnellement, ces adaptations sont traitées en temps d'interruption du système. La demande pour automatiser ce processus et pour le réaliser lors du fonctionnement du système est croissante. L'introduction des systèmes auto-adaptatifs était la réalisation d'un system en permanente adaptation. Les systèmes auto-adaptatifs peuvent modifier, au moment de l'exécution, leur comportement et / ou leur structure en fonction de leur perception de l'environnement, du système même et de leurs exigences. L'objectif de ce travail est de réaliser l'auto-configuration, une propriété clé et essentielle des systèmes auto-adaptatifs. L'auto-configuration est la capacité de se reconfigurer automatiquement et dynamiquement suite aux changements, tel que l'installation, l'intégration, le retrait et la composition / décomposition d'éléments du système.

Cette thèse présente le cadre du Dr-BIP, une extension du plan BIP pour la modélisation des systèmes à configuration automatique qui repose sur une approche basée sur un modèle et sur des composants et des connecteurs pour prescrire des systèmes. La combinaison de ces deux approches exploite les avantages de chacune d'elles, faisant de leur combinaison une méthodologie idéale pour la réalisation des systèmes auto-adaptifs complexes.

Un modèle de système Dr-BIP est un modèle d'exécution qui capture le système en cours d'exécution à trois niveaux d'abstraction différents, à savoir du comportement, de configuration et des variantes configurations. La configuration du système est saisie par des composants et des connecteurs. Dans un système de composants et de connecteurs, la configuration automatique (l'auto-configuration) peut avoir trois niveaux de granularité différents, notamment la possibilité d'ajouter ou de supprimer des connecteurs, d'ajouter ou de supprimer des composants et d'ajouter ou de supprimer des sous-systèmes. Dr-BIP prend en charge l'ajout et le retrait explicites de composants et de sous-systèmes, mais l'ajout et le retrait implicites de connecteurs. L'avantage principal de l'addition et de la suppression implicite de connecteurs est la capacité de garantir par construction une configuration spécifique de topologies.

Pour capturer les trois niveaux d'abstraction, nous introduisons des motifs en tant que structures principales pour prescrire un système Dr-BIP à configuration automatique. Un motif définit un ensemble de composants qui évoluent en fonction de règles d'interaction et de reconfiguration. Un système est composé de plusieurs motifs pouvant éventuellement partager des composants et évoluer ensemble. Les règles d'interaction dictent la manière dont les composants du système peuvent interagir, tandis que les règles de reconfiguration dictent l'évolution de la configuration du système. Enfin, nous montrons que le cadre proposé est à la fois minime et expressif en modélisant quatre systèmes différents à configuration automatique.

Finalement, nous proposons un langage de modélisation pour codifier les concepts du cadre et fournir une implémentation d'interprète.

Keywords: *reconfigurable, dynamic, self-configuring, self-adaptive, component and connector architectures*

Associated Papers

Several chapters in this thesis appeared in several papers in the form of conference articles or Verimag technical reports.

References

[1] *DR-BIP - Programming Dynamic Reconfigurable Systems*. Tech. rep. TR-2018-3. Verimag Research Report,

[2] Rim El Ballouli et al. "Four exercises in programming dynamic reconfigurable systems: methodology and solution in DR-BIP". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 304–320.

[3] Rim El Ballouli et al. "Programming Dynamic Reconfigurable Systems". In: *International Conference on Formal Aspects of Component Software*. Springer. 2018, pp. 118–136.

[4] *Four Exercises in Programming Dynamic Reconfigurable Systems: Methodology and Solution in DR-BIP*. Tech. rep. TR-2018-7. Verimag Research Report,

Contents

I Foundation and Preliminaries **1**

1 Introduction **2**
1.1 Motivation . . . 2
1.2 Self-adaptive Systems . . . 4
1.2.1 Definition . . . 4
1.2.2 Requirements . . . 5
1.2.3 The Self-* Properties . . . 8
1.2.4 Challenges . . . 8
1.3 Problem Statement . . . 12
1.4 Contributions . . . 14
1.5 Thesis Roadmap . . . 14

2 Existing Methodologies **16**
2.1 Approaches and Techniques . . . 17
2.1.1 Control Engineering . . . 17
2.1.2 Artificial Intelligence . . . 19
2.1.3 Software Programming . . . 19
2.1.4 Software Engineering . . . 21
2.2 Model-based X Component and Connector . . . 25

II Dr-BIP Framework **28**

3 Dr-BIP Framework **29**
3.1 Overview . . . 30
3.1.1 Design Pillars . . . 30
3.1.2 Conceptual Model . . . 31
3.2 Dr-BIP System Model . . . 34
3.2.1 Architecture Overview . . . 36
3.2.2 Components . . . 40
3.2.3 Motifs . . . 41
3.2.4 Motif-based Systems . . . 48
3.3 Dr-BIP as an Extension of BIP . . . 52
3.3.1 Component-based Systems . . . 52
3.3.2 Existing BIP Extensions for Dynamic Reconfiguration . . . 54

4 Dr-BIP by Examples **56**
4.1 Self-configuring Token Ring System . . . 56
4.2 Self-configuring Multicore Task System . . . 59
4.3 Autonomous Highway Traffic System . . . 63
4.4 Self-configuring Robot Colonies . . . 66

5 Implementation **71**
5.1 Overview . . . 71
5.2 Concrete Syntax . . . 72
5.2.1 Lexical Structure . . . 73
5.2.2 Grammar Highlights . . . 75
5.2.3 An Example Using The Concrete Syntax . . . 79
5.3 Parser . . . 80
5.4 Interpreter . . . 81
5.4.1 Parameters . . . 82
5.5 Execution . . . 82

III Conclusions and Perspectives **85**

6 Conclusion and Perspective **86**

A Appendices **95**
A.1 Self-adaptive System Definitions . . . 95

List of Figures

1.1 Conceptual model for a self-adaptive system: dissecting the basic principles of a self adaptive system 6
1.2 Adaptation loop describes the process adopted by the adaptation engine to achieve adaptability 7

2.1 A broad classification of approaches used in designing and developing self-adaptive systems 17
2.2 The basis of Dr-BIP approach is the combination of two approaches namely, the model-based and architecture-based approach 25

3.1 Conceptual model of Dr-BIP framework: dissecting the basic principles of a self-configuring system 32
3.2 Adaptation loop describes the process adopted by the Dr-BIP adaptation engine to achieve reconfiguration 34
3.3 Dr-BIP system model abstracts the target system at three different levels of abstraction 35
3.4 An example of reconfiguration in a motif-based system 37
3.5 Abstract syntax of interaction and reconfiguration rules 37
3.6 An example of a motif definition 38
3.7 Reconfiguration vs Interaction Steps 39
3.8 An example of a component type 41
3.9 Overview of motifs structure and evolution rules 42
3.10 An example of a motif type 44

3.11 An example of a set of multiparty interactions in a motif 46
3.12 An Overview of motif-based systems 49
3.13 An example of a static configuration with BIP 54

4.1 Self-configuring token ring system 57
4.2 Dynamic ring system evolution across 1,000 steps 59
4.3 Self-configuring token ring system's measurements 59
4.4 Self-configuring multicore task system 60
4.5 Task load across 3000 steps 62
4.6 Self-configuring multicore task system's measurements 63
4.7 Automated Highway Traffic System 64
4.8 Automated highway traffic evolution along 13 sampled steps 66
4.9 Automated highway traffic system's measurements 66
4.10 Self-organizing robot colonies 68
4.11 Reconfiguration rules of a self-configuring robot system 70

5.1 An overview of the prototype implementation 72
5.2 Partial view of an abstract syntax tree 84

List of Tables

1.1 Variations in the definition of self adaptive system in the first school of thought 5

2.1 List of advantages and disadvantages of control-based approach . . 18
2.2 List of advantages and disadvantages of artificial intelligence approach 20
2.3 List of advantages and disadvantages of software programming approach 21
2.4 List of advantages and disadvantages of component-based approach 22
2.5 List of advantages and disadvantages of model-based approach . . . 24

5.1 Lexical structure of the Dr-BIP language 74
5.2 Highlight of Dr-BIP grammar rules 76
5.3 Description of options available to cater the execution of the prototype 83

Part I

Foundation and Preliminaries

Chapter

1 Introduction

Contents

1.1 Motivation 2
1.2 Self-adaptive Systems 4
1.2.1 Definition 4
1.2.2 Requirements 5
1.2.3 The Self-* Properties 8
1.2.4 Challenges 8
1.3 Problem Statement 12
1.4 Contributions 14
1.5 Thesis Roadmap 14

1.1 Motivation

Software systems have invaded our lives in the past thirty years [1] and no downfall can be seen in this regard. In fact, we have become highly dependent on software systems in our day-to-day tasks and the demand is ever so increasing. For example, on a typical day one relies on multiple software systems: waking up on

a digital alarm clock, answering emails through a mobile phone, prepping a meal with home appliances, etc. The daily dependability on systems has impacted our expectations from software systems; expecting them to be energy-efficient, flexible, resilient, customizable, self-optimizing, etc.

The life cycle of a software system typically involves: analysis, design development, testing, and deployment. Once the system is deployed, it enters the *software evolution* phase by which it is maintained by a system administrator to handle faults, improve performance, address changes to meet changing requirements, etc. Such systems are known to have an *open-loop* structure and require external intervention to evolve. Evolution could be triggered by either *internal* factors that stem from the system itself such as failure, or *external* factors that stem from the system's environment such as change in requirements, or emergent of new technology that must be integrated. Systems that don't respond to change factors will progressively become less useful and hence software evolution is inevitable.

The evolution of some systems has become time consuming and a hassle even to the most skilled system administrator. This is caused by the exponential growth in size of such systems leading to a "complexity crisis" [2]. In addition, the cost of evolution has been steadily increasing and is estimated to be more than 90% of the total cost of the entire system's life cycle [3]. Therefore, the demand to achieve the desired requirements within a reasonable cost and time becomes apparent.

In addition to the "complexity crisis" hindering system's evolution, a special type of systems that belong to IOT (Internet Of Things) introduces new dimensions of complexity. IOT or Industry 4.0 enables heterogeneous embedded systems or objects in general to sense their surrounding and interact with each other through a communication network to achieve global goals. Some application examples of IOT include home or industrial automation, automotive traffic management, smart cities and many others. An enormous amount of sensor data is generated by such systems, which must be continuously analyzed in order to adapt to changes in either the context or environment whilst achieving system goals and requirements. Such systems must be context-aware as they are subject to unpredictable changes in context that cannot be anticipated before deployment. Moreover, such systems are expected to handle the adaptation at runtime as the need arises.

In summary, maintenance that is decoupled from the runtime environment and performed manually is difficult and expensive due to the size and complexity of systems. Furthermore, context-aware systems require adaptation to be handled at runtime. Therefore innovative ways are required to design, develop, and deploy such software systems.

On the one hand, *Autonomous systems* came along as a solution to minimize human intervention and reduce evolution cost in complex software systems. Autonomous systems "manage themselves given high-level objectives from adminis-

trators" [4]. In other words, such systems take a major load off system administrators as they are only required to dictate new objectives. kephart and Chess argue that autonomous systems are the rightful approach to tackle problems arising from the "complexity crisis" [4]. On the other hand, *self-adaptive systems* were introduced as a realization of continuously adapting context-aware systems. Self-adaptive systems are "able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their requirements" [5]. Furthermore, they handle adaptation and evolution at runtime. This can be achieved by converting an *open-loop* system to a *closed-loop* system with the aid of a feedback loop that adjusts the system at run-time.

Self-adaptive and autonomous systems are strongly related, and it is difficult to draw distinction between the two terminologies as they are used interchangeably in the literature. Self-adaptive systems are more specific and have less coverage than autonomous systems [6]. Consider a software system decomposed into the conventional layered model consisting of application, middleware, network, operating system, and hardware. In Self-adaptive systems adaptation covers only the application and middleware layer, however, in autonomous systems adaptation covers the application, middleware, network, and operating system layers.

1.2 Self-adaptive Systems

This section presents a general overview of the basic concepts in self-adaptive systems. First, it briefly discusses different interpretation produced by the research community for the term self-adaptive systems. Next, it elaborate upon the basic foundations, requirements and challenges of self-adaptive system. Finally, it introduces four properties that are oath to be characteristics of self-adaptive systems.

1.2.1 Definition

Self-adaptive systems are still the focus of intense research and development. There exists an enormous amount of literature contributing to the modeling, design and development of self-adaptive systems, however there is no consent on its definition.

One school of thought defines self-adaptive systems as those that adapt in response to change. There are slight variations within this school of thought. These differences arise from three questions: *what* can be modified/altered, *when* is the adaptation triggered (i.e. what are the monitored properties), *how* is the adaptation performed? In response to *what* can be modified, three different approaches exist either by modifying the system's behavior [7–10], or the system's structure

What	Structure	[5, 6, 11, 12]
	Behavior	[5, 6, 8–10, 12]
When	Environment	[5, 6, 8–15]
	Requirements	[5, 10–13, 15]
	System	[5, 6, 8–10, 12, 14, 15]
How	Run-time	[6, 10, 14]

Table 1.1: Variations in the definition of self adaptive system in the first school of thought

[11], or both [5, 6, 12]. The adaptation is triggered *when* a change is detected in the environment [8], requirements [13], system-itself [14] or a combination of these three [5, 12, 15]. When considering *how* adaptation takes place, some emphasize that it is handled at runt-time [6, 10, 14], while others don't. A detailed classification of variations within this school of thought is presented in Table 1.1.

Another school of thought defines self-adaptive systems as those whose main aim is to meet system requirements despite uncertainties or changes that may arise in operating conditions [7, 16–18]. In other words, such systems evaluate performance and whenever they are not accomplishing what they are intended to do, possibly due to failure or variability in resources, adaptation is triggered.

In comparison, the system requirements, in the first school of thought, can be variable and are subject to change, but are fixed in the second school of thought. Furthermore, any change in the system requirements triggers adaptation in first school of though, while unsatisfied requirements triggers adaptation in the second school of thought.

Henceforth in this dissertation we utilize the definition of self-adaptive systems given by [6] as it adheres closely to our proposal. In other words, a self-adaptive system is one that adapts at runtime to changes in itself and the environment.

1.2.2 Requirements

This section presents the requirements of a self-adaptive system with the aid of a generic conceptual model adopted by such systems. Furthermore, it describes the adaptation loop which highlights the fundamental modules required in a self-adaptive system to attain adaptability. Moreover, this section introduces a set of terminologies, that are used hereafter in the dissertation.

Conceptual Model

The conceptual model describes the abstract elements composing a self-adaptive system and the relation between them. In other words, it presents the basic prin-

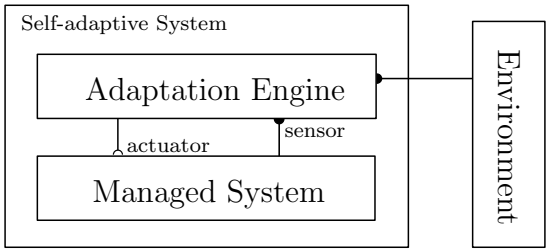


Figure 1.1: Conceptual model for a self-adaptive system: dissecting the basic principles of a self adaptive system

ciples of self-adaptive systems. The conceptual model is composed of three elements: the managed system, adaptation engine, and environment. Hence, a self adaptive system can be seen as a tuple $SAS = (MS, AE, E)$. Figure 1.1 depicts the anatomy of the conceptual model and a description of each entity is presented next.

Managed system. comprises the application code which realizes the functionality of the system. In the case of collaborative adaptive systems the managed system can be thought of as a series of resources such as robots, vehicles, etc. To support adaptation, the managed system is equipped with actuators. Actuators enable the execution of adaptation requests selected by the adaptation engine. For instance, given multiple robots that collaborate to transport an element from point A to B, the managed system is responsible for the navigation of robots and element transfer. The actuators may restrict five robots to participate in the transfer of the element depending on its weight. Different terms are used in the literature referring the concept of managed system. For example, it is also referred to as managed element [4], system layer [19], adaptable software [6], managed resources [20], base-level subsystem [21], and component control layer [11].

Adaptation Engine. supervises and administrates the managed system. It contains the adaptation logic needed to achieve system requirements or goals. The adaptation engine is equipped with sensors that monitor both the managed system and environment and adapts the preceding when necessary. The adaptation engine analyzes the monitored data and constructs an adaptation plan. For instance, consider a robot that adapts its navigation strategy depending on the presence of obstacles (sensed from environment) and its energy level (sensed from the managed system). Different terms are used in the literature referring to the concept of adaptation engine. For example, It is also referred to as autonomic manager [4], architecture layer [19], adaptation logic [20] and reflective subsystem [21].

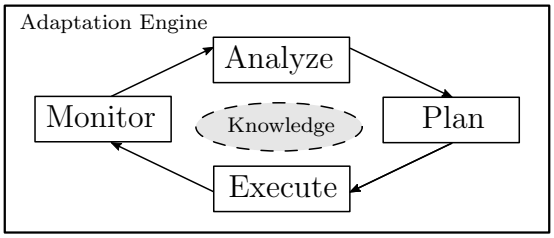


Figure 1.2: Adaptation loop describes the process adopted by the adaptation engine to achieve adaptability

Environment. refers to the external world with which the system interacts and is effected by. It includes physical entities such as obstacles on a robot's path.

In conclusion the conceptual model described above sheds light over self-adaptive systems with *external* adaptation approach i.e. having a clear separation between the adaptation engine and managed system. This separation increases maintainability through modularization and localization [6, 19, 22]. It is worth noting that other approaches exist where the adaptation engine and managed system are intertwined into a single unit. Such self-adaptive systems are known to have *Internal* adaptation approach. With the internal approach the sensors, actuators and adaptation logic are mixed with the application code, often leading to poor maintainability and scalability. Empirical evidence in favor of external adaptation over internal adaptation can be found in [23].

Adaptation Loop

As discussed in section 1.1, self-adaptive systems deploy a closed-loop mechanism, also known as adaptation loop [6]. The adaptation loop comprises the process used by the adaptation engine to achieve adaptability. It is inspired by the MAPE-K control loop in autonomic computing and it envelopes four steps, monitoring, analyzing, planing and executing [4]. The effectiveness of the MAPE-K comes from its intuitive structure in handling the different functions required for a feed-back loop [16]. Figure 1.2 illustrates the adaptation loop process.

The first step is to *monitor* and collect data from the environment and managed system through sensors. The collected data is processed and knowledge is updated. Next, up-to-date knowledge is *analyzed* to determine whether an adaptation is needed to attain system requirements or goals. Next, if adaptation is mandatory, a *Plan* is constructed consisting of one or more adaptation actions. Finally, the plan is *executed* by the managed system with the aid of actuators.

To summarize, a self-adaptive system requires sensors to monitor the environment and managed system. It also requires the presence of an adaptation engine which has the capability to monitor and collect data, analyze collected information,

and construct a convenient adaptation plan. finally, self-adaptive systems demand the presence of actuators which will aid the managed system in the execution of the adaptation plan.

1.2.3 The Self-* Properties

A self-adaptive system adapts at runtime to changes in itself and environment. To achieve this, ideally speaking, systems should have certain adaptive characteristics known as the *self-** properties. These properties are introduced in autonomic computing [2, 4] and have been here after referred to in the context of self-adaptation in many works [6, 16] as basis for adaptation. These properties are composed of four categories, which are discussed in detail next.

- *Self-configuring*: is the capability of reconfiguring automatically and dynamically in response to changes. This may include installing, integrating, removing and composing/decomposing system elements.
- *Self-optimizing*: is the capability of managing performance and resource allocation whilst satisfying user requirement. This includes concerns such as throughput, response time etc.
- *Self-healing*: is the capability of discovering, diagnosing, and reacting to disruptions. This include both reactive or proactive healing. In proactive healing potential problems are anticipated and acted upon early on to prevent failure. while self-repairing focuses on recovery from them.
- *Self-protecting*: is the capability of detecting security breaches and recovering from their effects. This includes both reactive and proactive protection, namely recovering from both existing attacks and anticipated ones.

While the majority of researchers in the field agree that self-adaptive systems are expected to embody all of these properties, only few researchers have directed their focus to more than a single property to aid with realizing self-adaptive systems, such as [19]. This is because of the difficulty of orchestrating and designing systems whilst keeping in mind all four properties. Henceforth in this dissertation the focus will be targeted towards the self-configuring property. In other words the focus of this dissertation is on *self-configuring adaptive systems*.

1.2.4 Challenges

Self-adaptive systems pose new challenges to the development and design of software systems. This section aims to identify the various challenges faced by

software engineers in realizing self-adaptive systems. It first tackles challenges in the broader view of the domain. Next it discusses challenges with respect to the requirements of a self-adaptive system presented in section 1.2.2, namely the conceptual model, adaptation loop, and self-* properties.

Framework Challenges. While there is a handful of research dealing with approaches to reason about realizing self-adaptive systems, there is a lack of language, tools and integrated frameworks that integrate and embody these concepts. Furthermore, existing frameworks such as Stitch [24] (a language to model repair strategies in an adaptive system) are usually domain specific and lack generality. A main challenge it to develop frameworks that are generic enough to tackle problems in various domain, yet expressive enough to model complex problems. Having a general purpose framework to realize self-adaptive systems facilitate the integration of these frameworks by the industry.

Trust Challenges. One of the main challenges faced after releasing a self-adaptive system into industry is lack of trust that users/administrators have when dealing with such systems. This mainly arises due to three problems. First, the self-dependence of the system leaves it untraceable and hence the user is left abandoned with regards to the actions that the system is choosing/performing. One way to solve this issue is to report the activities and decisions made by the self-adaptive system to the administrators. Determining how much information to expose and what are concise ways to represent such information remains a challenge to be addressed by system engineers. Second, the lack of user control over the self-adaptive system encourages users to neglect it. Deciding on how much control to delicate to users is a challenging question. For instance, when the system's and administrator's decision are conflicting, which action is overridden? and in which situation can an administrator override the system's decision? Third, there is no consent in the literature on a single metric to measure the quality of adaptation. Such a metric will convey confidence and encourage the acceptance of self-adaptive systems. To ease the adoption of self-adaptive systems into industry these trust challenges needs to be addressed.

Conceptual Model Challenges

Adaptation Engine Challenges. While most existing work focuses on a centralized adaptation engine, few works started addressing decentralization such as [17]. The authors in [17] investigate the different patterns in decentralizing the adaptation loop which comprises the main functionalities in the adaptation engine. Decentralization and distribution of the adaptation engine is inevitable when dealing with complex and scalable systems. Decentralization and distribution bring

in new challenges to the table. First, they introduce the need for effective communication protocols to share knowledge across the adaptation engines. Another issue to consider is latency that might be introduced by communication protocols. Latency results in temporal inconsistent views of the system. The main challenge is to develop algorithms that supervise and administrate the managed systems while tolerating inconsistency.

Managed System Challenges. The managed system is typically modeled into a representation that reflects the actual systems behavior. The key issue when modeling a system is picking the right level of abstraction. How much information to abstract away? If the model is too abstract, it may be easier to control by the adaptation engine, however it may no longer reflect the actual system. On the other hand, complex detailed models are difficult to deal with from the adaptation engine's perspective. Therefore, the consistency between the model and the managed system must be maintained and this challenging trade off must be taken into consideration by engineers when designing the managed system model.

Sensor & Actuator Challenges. The sensors and actuators are mainly used by the adaptation engine to peak into what is happening in the managed system and to accordingly make changes to it. One key challenge is deciding on what can be sensed i.e. the exact information needed to make precise and correct adaptation decision. Other important questions to answer are what actuators are needed to change the system? Which architecture styles support both sensing and acting? The sensors and actuators are usually catered to the system's goal. In other words the system goal determines what information is needed and what can be modified in the managed system to reach the goal. A major challenge is addressing goal change and accounting for new sensors and actuators that might be needed to achieve new goals.

Environment Challenges. To capture uncertainty in the environment it must be modeled. Existing work captures modeling of parametric uncertainties where the value of a certain element in the model is unknown. Some challenging questions include: How to deal with more complex uncertainties? How to deal with real life uncertainties whose behavior can't be completely captured and translated to a model? One possible solution is to rely on discrepancy modeling. Moreover, It is also important to think ahead about how to deal with new uncertainties that might have not been modeled.

Adaptation Loop Challenges

Monitoring Challenges. The objective of the monitor is to capture and collect sensed data. The information being monitored and gathered is usually determined by the system goal. Gathering and collecting all the information from sensors is very costly. Furthermore, monitoring for multiple goals may lead to redundant information and consequently undesirable costs. Hence a challenge to consider is the tailoring of the monitor depending on the situation being analyzed and system goal. Finally, majority of existing approaches determine in advance (and at design time) what to be monitored, however the main challenge is to have adaptive monitoring, where the monitoring process is updated to account for new emerging system goals.

Analysis Challenges. Given the monitored information, the main goal of the analysis is to determine when the system is in a bad state. A bad state typically refers to an undesired system behavior which requires adaptation. How well it can detect a bad state and will it be detected soon enough to take proper actions? These are some of many investigations that should be addressed. The analysis task to this date is considered a major challenge. In-fact its complexity has lead researchers to rely on ad-hoc and rule-based techniques for analysis. Promising approaches are the use of artificial intelligence and data mining techniques to adopt on-line analysis.

Planning Challenges. The planer takes a screen shot of the system current state along with the system goals to decided on an adaptation plan that satisfies the system constraints and goals. The adaptation plan is a sequence of actions that must take the system from an undesired state to a normal state. Unfortunately, This task is computationally hard and as such, most researchers rely on off-line planning. In off-line planning a set of plans are created at design time that can be shown either by construction or by a verification process to satisfy system constraints. However, the real research challenge lies in on-line planing, where new plans are synthesized on the fly as the system goals change. Other challenges include dealing with planning for multiple goals and conflict resolution, accounting for incomplete system information in decentralized systems, and insuring the planned transient behavior is safe.

Execution Challenges. At this stage the managed system executes the adaptation plan with the aid of actuators. Matters to consider at this stage are: how to handle the failure of completing the execution of the adaptation plan and the interference between the execution of multiple adaptation plans. In addition, an important step is to validate that the execution of the adaptation plan in-fact is

correct and results in the desired behavior. Most of existing approaches rely on limited examples to show the validity of their approach, however verification is an essential step. The adaptive behavior of such systems dimensions the need for static verification and strengths the need for runtime verification. Relying on runtime verification for adaptive system is complex due to the several alternatives and execution paths that are inherent to the nature of self-adaptive systems. A combination of both off-line and on-line verification seems a possible, but challenging, resolution.

Self-* Property Challenges

Since the focus of this dissertation is on the self-configuring property, this section tackles only the challenges that are encountered in realizing self-configuring adaptive systems. A self-configuring adaptive system is one which allows the installation, integration, removal and composition/decomposition of system elements at runtime in response to changes that arise in its environment or itself.

The research challenges are primarily concerned with transient behavior. It is not only important to make sure that transient behavior is of desirable characteristics but also that the system safety property is not violated during reconfiguration. In addition, it is essential to advocate seamless integration of new elements introduced to the system. An associated challenge is to verify that the new configuration in-fact satisfies system constraints. Another crucial point to address is making sure state information is not lost when configuration is modified. One possibility to approach this is by making sure that involved system elements are idle when performing a reconfiguration.

Finally, there are two interesting issues that arise as a result of large and complex self-configuring systems. First, in complex system its highly likely to have multiple elements exposing the same behavior. It also possible that you would like to introduce an element having the exact same behavior. For example, introducing new servers to address high user demands. This introduces the need for some way to capture behavioral types which allows the creation of several elements of a certain type (i.e. behavior). Another fundamental aspect to address in complex systems are shared elements, more specifically how to handle reconfiguration of shared elements (i.e. coexisting in multiple subsystems). All these are interesting complications that emerge as a result of reconfiguring systems.

1.3 Problem Statement

Modern systems are pressured to adapt in response to their constantly changing environment to remain useful. While traditionally, this adaptation has been

handled manually and at down times of the system by system administrators, there is an increased demand to automate this process and achieve it whilst the system is running. This is partly because manual adaptation of system has been estimated to cost more than 90% of the total cost of the entire system's life cycle [3].

For instance, consider the integration of an extra server replica to a web-based system to handle the overload in user demands. Also consider the removal of a faulty system element that is causing undesirable behavior and its integration later on after it has been fixed. Consider the removal of an entire subsystem that is reliant on an old technology and replacing it with one that integrates new a technology. More concrete examples can be found in systems that belong to IOT (Internet Of Things). For instance, consider an automated highway system where a bunch of cars are constantly entering the highway, communicating with each other in such a way to avoid traffic congesting and to optimize car flow. As each car reaches the end of the highway it leaves the system. In this way a bunch of cars are constantly entering and leaving the system. In IOT this behavior is intrinsic as constantly new devices are being introduced and handling such an adaption manually is not a practical solution.

All of the above examples involve a special type of adaptation, namely reconfiguration. In-fact self-configuration is one of four key attributes (the self-*attributes in section 1.2.2) intrinsic to self-adaptive systems. Briefly, a system configuration denotes the composed set of system elements and the connections among them and a reconfiguration implies the integration, removal, composition or decomposition of system elements. The focus of this dissertation is directed towards reconfiguration i.e. self-configuring adaptive system. A self-configuring adaptive system is one which allows the installation, integration, removal and composition or decomposition of system elements at runtime in response to changes that arise in its environment or itself.

This dissertation introduces Dr-BIP a formal framework for modeling self-configuring systems that relies on an architecture-based approach. An architecture based approach provides an appropriate level of abstraction to describe dynamic change in a system. Furthermore, architectures are scalable and hence they facilitate the description of large-scale complex systems. We introduce motifs as the architecture basis to structure the system and to coordinate its reconfiguration at runtime. An architectural motif defines a set of components that evolve according to interaction and reconfiguration rules. A system is composed of multiple motifs that possibly share elements and evolve together. Interaction rules dictate how elements composing the system can interact. Reconfiguration rules dictate how the system configuration can evolve over time. The dissertation lays down the formal foundation of Dr-BIP, implementation and illustrates its expressiveness on

several examples.

1.4 Contributions

This thesis presents Dynamic Reconfigurable BIP (Dr-BIP), a model-based approach that covers the specification, and execution of self-configuring adaptive systems. The main contributions are as follows:

- *Generality.* Proposal of a general framework that relies on common and effective architecture concepts making it applicable to a wide range of domains. Therefore, system engineers will not require any specific domain knowledge to specify self-configuring systems in Dr-BIP.
- *Gurantee by construction.* Definition of architectures as parametric operators on components guaranteeing by construction specific structural/functional properties.
- *Semantics.* Providing a sound foundation for analysis and implementation through the definition of formal and rigorous operational semantics in the form of state transition system. The semantics leverage on existing static BIP semantics (for component-based systems). A Dr-BIP system can be seen as a static BIP system as long as it is not executing a reconfiguration.
- *Separation of concerns.* Keeping separate the system's functionality from its self-configuring behavior. This avoids as much as possible blurring the behavior of components with information about their execution context and/or reconfiguration needs and thus enable reasoning about the system's adaptive behavior in separation of its functional behavior.
- *Coverage.* Demonstration of the framework coverage with four example coming from various domains including autonomous systems. We show that the framework is minimal, reusable and expressive allowing concise modeling.
- *Integrated.* Definition of a modeling language to accompany the framework concepts and provisioning of a packaged tool set which includes an interpreter for the language.

1.5 Thesis Roadmap

The thesis is organized into three main parts. The first part, introduces *what* makes up self-adaptive by presenting it's requirements, properties and challenges.

Next, it focuses on *how* self-adaptive systems are engineered by providing an overview of approaches and techniques while detailing the advantage and disadvantage of each. Furthermore, it expands on a single approach which is the backbone and basis of this dissertation, namely the model-based component and connector architectural approach, by presenting its details and advantageous.

The second part introduces the Dr-BIP framework by detailing, its design pillars, conceptual model, and its architectural elements that are used to compose a self-configuring Dr-BIP system. Next, it highlights the relation between Dr-BIP and its predecessor BIP and discusses existing extensions of BIP supporting self-configuration. After which it presents four examples of self-configuring systems modeled in Dr-BIP. Each example is first introduced with an explanation of the intended target system's behavior and then its modeling using Dr-BIP. Last but not least, it presents the prototype implementation of the Dr-BIP framework which includes a concrete syntax to describe motif-based systems, a parser and an interpreter for the operational semantics.

The third part wraps up the dissertation with a summary of key points from part 1 and 2 along with possible extensions and future perspectives.

Chapter

2

Existing Methodologies

Contents

2.1 Approaches and Techniques 17

2.1.1 Control Engineering 17

2.1.2 Artificial Intelligence 19

2.1.3 Software Programming 19

2.1.4 Software Engineering 21

2.2 Model-based X Component and Connector 25

In the previous chapter, the focus was on *what* makes up self-adaptive system, along with it's requirements and challenges. This chapter focuses on *how* self-adaptive systems are engineered by providing an overview of approaches and techniques. It details the advantages and disadvantages of each approach along with a few example references. Next, it expands on a single approach which is the backbone and basis of our work, namely the model-based component and connector architectural approach.

2.1 Approaches and Techniques

Extensive efforts have been put by engineers and researchers from different disciplines to realize self-adaptive systems. This section discuses various approaches that have been developed over time. Each approach is inspired by a specific discipline and as such highlights complementary aspects of realizing self-adaptive systems. Figure 2.1 lists the different disciplines and the several approaches branching from each discipline. For instance, developing a self-adaptive system from a control engineering perspective implies designing a control-based self-adaptive system whose behavior can change according to a set of well-formed mathematical models that can be formally analyzed.

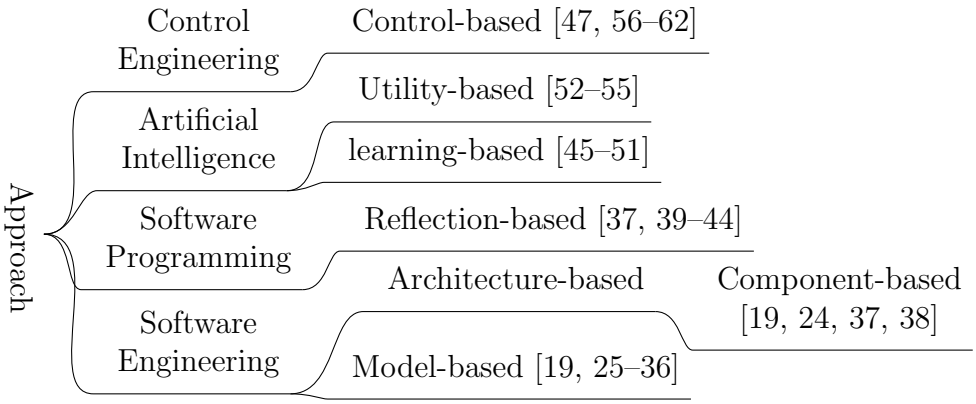


Figure 2.1: A broad classification of approaches used in designing and developing self-adaptive systems

2.1.1 Control Engineering

Control engineering is a discipline whose focus is on designing systems that behave as expected with the aid of system controllers. Traditionally, control engineering has been concerned with systems that are governed by the laws of physics, such as physical control plants. In recent years, the application of control theory to computing context have been studied in various works [63–65]. Furthermore, the similarities between physical control plants and self-adaptive systems are evident. Physical plants are constantly reacting to their environment to reach a certain goal and so are self-adaptive systems.

In this approach there are two fundamental concepts, namely the *target system* and *controller*. The controller implements a control strategy that dictates the correct control signal which adapts the target system in order to maintain the output of the target system sufficiently close to the desired goal. The control

signal is typically based on the difference between the previous target system's output and the system's goal. The target system is an analytical model based on mathematical relationships that relate the effect of the control signal on the system's behavior. For more details on the different techniques used to design target systems and controllers in self-adaptive systems refer to [66, 67].

Advantages
- Provides a formal approach to design systems. - Has mathematical grounding that enables formal guarantees on the behavior of the controlled system including four main properties (convergence, robustness, stability, absence of overshoot). - Facilitates formal analysis and verification of nonfunctional properties of the system.
Disadvantages
- Requires control experience as applying of-the-shelf control theories will lack rigorous assessment of the adequacy of the chosen control strategy. - Requires a profound mathematical background to understand how to model the target system and to decide on the right level of abstraction in such a way to expose the needed behavior of the system without complicating the synthesis of the controller. - Translation from system design which is typically based on mathematical formulas into an implementation is non-trivial process and if not done properly, properties that are guaranteed at design might be lost through the process.

Table 2.1: List of advantages and disadvantages of control-based approach

A control-based system relies on a control loop to incorporate target system's output and outside disturbances. One prominent technique for organizing a control loop in self-adaptive systems is the MAPE-K loop which has been referred to as adaption loop in Section 1.2.2. In-fact many works emphasize the importance and application of MAPE-K loop in control-based self-adaptive systems [6, 16, 17, 20, 62]. A detailed description of different patterns to decentralize MAPE control loop in self-adaptive systems can be found in [17]. Examples of control-based self-adaptive systems include [47, 56–62]. The works [58, 62] tackle the application of control theory to design self-adaptive systems. The works [56, 57] focus on control strategies for self-adaptive systems with multiple goals. The works [47, 59] rely on a model-based representation of the target system to design of self-adaptive systems. Finally, the works [60, 61] utilize a supervisory control strategy over self-adaptive systems. A list of the main advantages and disadvantages of the control-based approach is highlighted in Table 2.1

2.1.2 Artificial Intelligence

AI provides the ability for systems to learn, improve and make decisions in order to perform complex tasks. The field of AI is broad and ranging from natural language processing, multi-agent systems, machine learning, utility theory among others. Self-adaptive systems have common grounds with artificial intelligence, namely dealing with unexpected scenarios. Referencing the adaption loop in section 1.2.2, which is a basic requirement of self-adaptive systems, artificial intelligence can be found useful in two main elements in the adaption loop, namely the analyze and plan element. AI techniques can play a central role in self-adaptation by processing large amounts of data and performing analysis and decision making. Artificial intelligence learning techniques can be used to better analyze and identify patterns in sensed data from the environment. Furthermore it can be used to make better decisions on the adaptation plan to be executed by learning from previous experiences. Artificial intelligence can't be thought of as a unique solution to realize self-adaptive system, but a supporting solution whom together with other approaches such as control-based in [47, 48] and component-based in [51] results in compelling self-adaptive systems.

Utility-based. Utility theory is another profound concept in artificial intelligence. Utility refers to "the quality of being useful" [68]. Utility theory deals with assigning a utility value for each possible outcome and choosing the best possible outcome based on maximizing the utility value. For example, the works [52–55] employ a utility function to optimize dynamic reconfiguration of resources in autonomic systems. A list of advantages and disadvantages of artificial intelligence to realize self-adaptive systems is highlighted in Table 2.2

Learning-based. Learning algorithms in artificial intelligence such as reinforcement learning [69] and genetic algorithm [70] can be incorporated in the planning phase by the adaptation engine of a self-adaptive system. In-fact the use of various learning algorithms to realize self-adaptive systems can be found in [45–51]. In the works [45, 46] reinforcement learning is used in autonomic computing. In the works [47–49] on-line learning models are used to realize self-adaptive behavior. Finally the works [50, 51] investigate collaborative learning in self-adaptive systems.

2.1.3 Software Programming

In this approach, general purpose programming languages are utilized to realize self-adaptive systems. One of the main techniques is known as reflection-based and is explained in details next.

Advantages
- Enhance the analysis of sensed data from the environemnt and planning from past experiences - Can be used in combination of other approaches to produce compelling adaptive systems
Disadvantages
- Evaluation of the system is necessary due to the heavy reliance on heuristics and probability in this approach - Use of Artificial intelligence may result in unpredicted behavior of the system and lack of behavioral guarantees

Table 2.2: List of advantages and disadvantages of artificial intelligence approach

Reflection-based. Reflection has been introduced to programming language community with the aim to increase programming flexibility and to allow the development of closed software systems, which do not require external interference. A reflective software system is one which has the ability to examine and modify both its behavior and structure. A programming language supporting reflection provides a number of features available at runtime that aid with reflection such as the creation of new class types at runtime, and instantiation of objects of classes that where not defined at compile-time. Many general purpose programming languages already posses reflective abilities such as JAVA, and C#. The ability of software to adapt itself is an intrinsic characteristic of self-adaptive systems.

Some examples of realizing self-adaptive systems through the use of reflection include [37, 39–44]. In [39] the authors rely on architectural reflection to realize self-adaptation. Architectural reflection is the ability of software system to adapt its structure including components, interconnections, and data types. On the other hand, the authors in [41] rely on behavioral reflection. Behavioral reflection is the ability of the software system to change its behavior including communication mechanism, algorithms etc. The extension of the concept of reflection to requirements realize self-adaptive system was proposed by [40]. They claim that a self-adaptive system should be requirements-aware. A requirements-aware system should be able to observe and react to its requirements in the same way it does for its structure and behavior.

One of the main disadvantages of the all of the above mentioned work is that they are platform specific solutions and complex software may be deployed on heterogeneous hardware, operating systems, etc. This has led to the emergence of reflective middleware. Middleware sit between the application and the underlying operating system and hence provide a level of platform independence. This provides considerable benefits in terms of interoperability and portability of dis-

tributed system services and applications. Plastik [42] is an example of the use of reflective middleware to capture self-adaptive systems. It integrates OpenCOM component model and the ACME/ARMANI ADL [37, 44]. Moreover the authors in [43] use reflective middleware technology to support self-healing systems using Open ORB.

A summary of advantages and disadvantages of software programming approaches to realize self-adaptive system can be found in Table 2.3.

Advantages
- Reflection can be a quick and easy fix to add adaptive behavior for small noncritical systems
Disadvantages
- Reflection inevitably induces additional performance overhead - Reflection provides unlimited access to the software implementation and this can lead to changes that affect the integrity of system if not dealt with care

Table 2.3: List of advantages and disadvantages of software programming approach

2.1.4 Software Engineering

Numerous research areas under software engineering have tackled the realization of self-adaptive systems. This section sheds light over two such approaches, one of which is the model-based approach. The model-based approach emerges from the Model Driven Engineering (MDE) discipline. In MDE, models are treated as primary entities to design, develop and implement software systems. The second approach is the architecture-based approach, where architectures are primary entities of description. System architectures represent systems using the high-level elements from which they are made. This can be done in various ways, one of which relies on components and connectors, also known as the component-based approach. We discuss both approaches in detail next.

Component-based. In a component-based architecture the system description is composed of components that encapsulate the system's functionality and connectors that dictates the interaction between components. Connectors relate one component to another usually through relationships such as data flow or control flow.

A component and connector architecture description can aid in the construction of self-adaptive by allowing the system to keep track of its structure. In other words, it prompts structural self-awareness, which is specifically important

to capture self-configuration behavior. Many component-based approaches represent the architecture in the form of model and propose a component and connector description language to accompany the such as STITCH [24], ACME [37], and COMMUNITY [38]. A comparison of these description languages among others can be found in [71]. The Rainbow framework [19] is a well-known component-based framework for self-adaptive systems. It relies on component architecture models to both monitor and adapt the system. It enables system designers to self-adaptation capabilities to systems in a cost-effective manner by providing reusable framework elements also known as architecture styles. A list of advantages and disadvantages of component-based approaches is highlighted in Table 2.4.

Advantages
- Abstraction that captures the system structure and facilitates the description of reconfiguration - Encapsulation is a key advantage which supports the separation of concerns - Component and connector architectures generally facilitate the description large-scale complex systems - component and connectors are common abstract concepts that can be used to describe self-configuring systems from various domains
Disadvantages
- Component-based architectures are favored for reusability, however the of level reusability is finely grained since components are not likely to be reused across systems from different domains - The system functionality is typically divided across component, which introduces dependability amongst component. Therefore, if any single component fails, then the entire system is affected. This introduces the need for self-configuration especifically in component-based systems.

Table 2.4: List of advantages and disadvantages of component-based approach

Model-based. A model is a representation of the system at some level of abstractions. A model can represent the system's requirement, architecture, implementation, or development, depending on the concern at hand the model captures only relevant information with respect to the model concern. Other type of models encapsulate nonfunctional properties of a system such as performance, tolerance, and security etc. A model is described using a modeling language, which is typically composed of *abstract syntax*, *concrete syntax*, and *semantics*. The abstract syntax describes the concepts of the language and their composition to create a model. The concrete syntax is a textual or graphical notation used to describe a model. The semantics employ the meaning of the language i.e. the interpre-

tation of a model written in the corresponding language. The semantics can be either defined formally using mathematical notations or informally using natural language.

Models can be used in two different ways, either as *development models* or *runtime models*. Development models start from the abstract model describing the system's requirements which is then systematically transformed and refined to architecture, design, implementation and deployment model until reaching the final running system. In other words, development models bridge the gap between the problem space and solution space where the problem space is the application domain and the solution space is the domain of implementation. However due to changing conditions in systems environment and insufficient information at design time relying on development models is not sufficient to realize complex systems, especially self-adaptive systems which are constantly adapting to the environment or their-self. This has lead to the extension of MDE to runtime, such models are known as models@runtime or runtime models.

A runtime model is a casually connected representation of system's structure, behavior or goals. A model is said to be causally connected to a running system if it is linked in such a way that if either the model or running system changes, this leads to a corresponding effect on the other. In other words, a casual connection is established in a bidirectional manner with the running system. In the first direction, the runtime model is kept up to date with the running system i.e. the model is an exact reflection of the running system at all times. In the opposing direction, the connected model can be used to effect change in the running system i.e. a change triggered at the model level is an equivalent change at the running system level. The primary advantage of Runtime models enable automatic monitoring and analysis of the system whilst its running. A complete list of advantages and disadvantages of model-based approach to realize self-adaptive systems is highlighted in Table 2.5.

The model-based approach have been sufficiently studied in the context of realizing self-adaptive systems [19, 25–36]. Some researchers rely on architecture models to represent the system [19]. An architecture model captures the structural architecture of the system in various representations including components or layers. Others rely on feature models to represent the system [25–27, 34]. A feature model captures potential variants of the system. Feature models offer a way of reasoning about adaptation by representing all possible configurations of self-adaptive system. Goal models can also be used to realize self-adaptive systems as in [28, 29]. A goal model captures system's requirements, once these requirements are not fulfilled an adaptation is triggered. It is often the case that a single runtime model is not enough to represent complex system and as such some researchers rely on multiple models at runtime. For example, a combination of both feature and

Advantages
- Models are modular and abstract representations of the system making them easier to handle and maintain than a system's actual implementation, this is especially relevant in large complex systems - Models can be used to automate the construction of a system implementation and to automate verification a system at runtime - Models are platform independent, and as such reduce both cost and time that may be needed to target various specific platforms
Disadvantages
- When multiple models are used at runtime it is challenging to maintain explicit relations across conflicting models making analysis and reasoning demanding - A model-based approach must be accompanied by other approaches for complete realizing of self-adaptive systems - Models if not accompanied with formal semantics can not be used for analysis or reasoning about the system

Table 2.5: List of advantages and disadvantages of model-based approach

architecture models to realize self-adaptation can be seen in [30]. Moreover, a combination of both architecture and behavior models to realize self-adaptation can be seen in [31, 32]. When utilizing multiple runtime models it becomes essential to maintain the relation between these models especially in the case of conflicts and overlap. For example, an adaptation triggered by one runtime model may violate constraints in another model. It is important to make relations between multiple runtime models explicit as to facilitate automatic analysis and reasoning. In fact, [33] proposes Euroma, a megamodel language to manage multiple models and their relation at runtime in self-adaptive system.

Similar to the AI approach, the model-based approach can't be considered alone as a solution for realizing self-adaptation in systems, but rather an assisting approach that needs to be accompanied with additional approaches in-order to support change in the running system. For example, a model-based approach can be accompanied with aspect-oriented approach [34], service-oriented approach [35], architecture-based approach [19], or component-based approach [36] to realize self-adaptive systems.

This list of approaches is meant to shed light on the variability of methods available in realizing self-adaptive systems and is not by any means exhaustive or complete. In fact there are other approaches such as the *agent-based* approach which emerges from the artificial intelligence discipline. In addition, several other alternative approaches emerge from the network computing discipline. We refer the interested reader to the following surveys [6, 16, 20, 72] for more information

on other omitted approaches.

2.2 Model-based X Component and Connector

The previous section provided a broad view on the several tracks that can be used to realize self-adaptive systems. Note that most of the techniques mentioned are used in combination with each other to achieve self-adaptation. For instance, [37] relies on both a component-based and a reflective-based approach to achieve self-configuration. This section aims to introduce the backbone approach of the Dr-BIP framework, which involves the interplay of two approaches previously discussed. More precisely, this section will present the reasoning and the advantages behind the Dr-BIP approach.

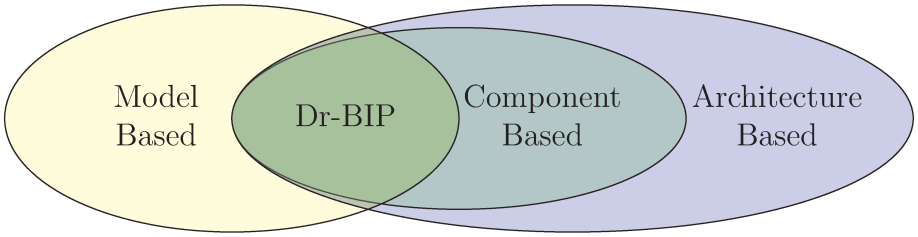


Figure 2.2: The basis of Dr-BIP approach is the combination of two approaches namely, the model-based and architecture-based approach

The basis of Dr-BIP is composed of the interplay of two approaches namely the model-based and architecture-based approach which are presented in section 2.1. The combination of both of these approaches exploits the benefits of each, making their combination an effective method to design and implement complex self-configuring systems. Figure 2.2 graphically displays the decomposition of the approach adopted by Dr-BIP.

An architecture-based approach to system design is one in which a system is represented in terms of its high-level elements composing it. Architectures abstract away from the complexity of the design, implementation, and deployment of systems. They provide an appropriate level of abstraction to describe reconfiguration in a system, such as the use of bindings and composition. They aid in shifting th the focus from the implementation details to the design of the entire system. Such an approach not only captures the system's functionality, but also the variability in the system configuration. Architectures and their underlying concepts and principles are generic, consequently making them applicable to a wide range of domains. Moreover, architectures generally support hierarchical composition which is useful for varying the level of description and thereby facilitating their use to realize large-scale complex systems. Systems based on architectures are easier to maintain

and upgrade. To summarize an architecture-based approach endorse *generality, abstraction, scalability, maintainability, and adaptability* of systems making it an ideal choice for modeling complex self-configuring systems.

There are multiple types of architectural elements that maybe considered when describing system architectures such as services, components, etc. The most common and fundamental architectural element to consider as base for architectural description is a component. In a component-based architecture, the system description is composed of components and connectors. Components encapsulate functionality of the systems and connectors dictates the interaction between components i.e. it relates one component to another usually through relationships such as data flow or control flow. One of the main benefits of component and connector architectures is encapsulation. It supports separation of concerns by keeping separate the system behavior (functionality) from the system architecture (the interaction between components). This allows the separation of the adaptive behavior from the non-adaptive one making the system easier to specify and more emendable to automated analysis. For example, components do not need to know under which interaction patterns they will be used, as long as their local interaction constraints are satisfied. Another benefit of component and connector architectures is reusability, as they allow one to specify the general case of an interaction pattern (connector) or a component behavior and reuse it. To summarize a component-based approach to architecture design endorses *encapsulation, and reusability*. It is for these reasons that Dr-BIP relies on component and connector architectural description.

In addition to the component and connector architecture approach, Dr-BIP also utilizes a model-based approach. In a model-based approach, a runtime model, which is a representation of the system, is casually connected to the actual running system facilitating runtime reconfiguration. More details on this approach can be found in section 2.1. An instance of a model-based approach aids in its adoption by the industry. This is because models are generic and platform independent. Moreover, these models can be automatically translated into general purpose programming languages) to target different platforms or devices, which reduces both cost and time required for system design and development. In addition, since models are representations of the running systems they enable system monitoring and hence offer early predictions about system's behaviors and properties. By formalizing models and clarifying the formalisms used, a model-based approach not only facilitate monitoring of the system, but also automatic analysis and verification of the system during operation. To summarize a runtime model-based approach endorses *industrialization, system monitoring, and automatic reasoning*

Component and connector architecture description languages (C&C ADLs) is one solution that combines both component-based and model-based approaches

to enable composition of system's architectural models from component. A C&C ADL is a formalism which is used to describe system architectures based on components and connectors. Tens of formal C&C ADLs have been proposed, each characterized by different conceptual architectural elements, syntax and semantics. Some C&C ADLs support only static configuration such as ArchFace [73], C3 [74], COSA [75], MontiArc [76]. In a static configuration, the system configuration is known at design time and is fixed through out its execution. Others support dynamic configuration i.e. self-configuration, such as ACME [37], RAINBOW [19], Dynamic Wright [77]. In a dynamic configuration the system is changing dynamically at runtime. Several surveys present a detailed comparison of C&C ADLs supporting dynamic reconfiguration [78–80].

Dr-BIP proposes a formal C&C ADL to aid in the modeling of self-configuring adaptive systems. The Dr-BIP framework relies on key concept of architectural motif as the elementary unit of description of self-configuring systems. A motif encapsulates (i) behavior, as a set of components, (ii) interaction rules between components (i.e. connectors) and (iii) reconfiguration rules about creating/deleting or moving components. Systems are constructed as a superposition of several motifs, possibly sharing their components, and evolving altogether.

In summary, Dr-BIP relies on an architecture-based approach. More precisely, it relies on a component & connector architecture to mitigate away from the complexity of system design. It also relies on a model-based approach, where by it proposes a formal C&C ADL to ease the modeling of self-configuring systems. The combination of these two approaches exploits the benefits of each.

Part II

Dr-BIP Framework

Chapter

3

Dr-BIP Framework

Contents

3.1 Overview 30
3.1.1 Design Pillars 30
3.1.2 Conceptual Model 31
3.2 Dr-BIP System Model 34
3.2.1 Architecture Overview 36
3.2.2 Components 40
3.2.3 Motifs 41
3.2.4 Motif-based Systems 48
3.3 Dr-BIP as an Extension of BIP 52
3.3.1 Component-based Systems 52
3.3.2 Existing BIP Extensions for Dynamic Reconfiguration . 54

The previous chapter, surveyed existing methodologies in the literature for realizing self-configuration. It further affirmed the need for a general integrated framework that is applicable to different self-configuration problems from various domains. The integrated framework aims to aid system engineers to not only design self-configuring systems, but also to monitor and reason about them. It

aims to help system engineers to add self-configuration abilities to systems in a cost effective manner that saves engineers both time and effort.

To accomplish this, we introduce Dr-BIP, an integrated framework accompanied with a language and an interpreter that codifies its concepts. The underlying principle is to maintain the separation between the system's functionality and its adaptive behavior (i.e self-configuration). To achieve this, Dr-BIP respects a strict separation between component behavior and its coordination. This separation is crucial to facilitate maintainability, and more importantly to reason about and analyze the system's adaptive behavior in separation of its functional behavior.

This chapter introduces Dr-BIP framework by first detailing its design pillars and conceptual model. Next, it introduce the Dr-BIP runtime system model by first presenting the architectural elements that can be used to compose a Dr-BIP model and then describes how they can be composed to model a self-configuring system. Finally, it highlights the relation between Dr-BIP and its predecessor BIP and discusses existing extensions of BIP supporting reconfiguration.

3.1 Overview

This section provides an overview of the Dr-BIP framework. It first introduces the foundation principles behind the framework. Next, it presents the conceptual model of Dr-BIP and relates it to the general self-adaptive system conceptual model introduced in section 1.2.2. Finally, it exposes the various processes embodied in the Dr-BIP adaptation engine that form an adaptation loop and relates it to the general adaption loop introduced in section 1.2.2.

3.1.1 Design Pillars

The Dr-BIP framework aims to cover, as much as possible, the current needs in the design of self-configuring systems. The Dr-BIP framework is built on concrete foundation pillars that are explained in details next.

General. A general framework allows to model problems of various complexities and problems coming from different domains. In other words, it enhances the coverage of problems that can be modeled and designed with such a framework. Moreover, system engineers utilizing a general framework will not require very specific domain knowledge to design systems. Dr-BIP endorses generality by relying on common, but effective, architecture abstractions such as component and connectors to model the system. Components capture the system's functionality and connectors capture multi-party interactions between components.

Rigorous. A rigorous framework provides sound foundation for analyzing and implementing the system. Dr-BIP relies on a well-defined operational semantics, leveraging on existing models (from its predecessor BIP) for rigorous component-based design.

Separation of concerns. A framework supports separation of concerns if it separates the system's behavior (functionality) from the system's adaptive behavior (self-configuration). Such a separation helps to avoid blurring the behavior of components with information about their execution context and/or reconfiguration needs. Dr-BIP achieves this by using exogenous global coordination rules which allows to reason separately about the system function and its adaptive behavior.

Support Runtime Models. A framework supports models at runtime by maintaining an abstraction of the target system that is casually connected to the running system. Runtime models facilitates to monitor, adapt, and reason about the system whilst its running. Dr-BIP support runtime system models which capture the target system at three different levels of abstraction. They capture behavior, configuration and possible configuration variants of the system. The Dr-BIP model is like a living concept, that can be updated at runtime using dedicated primitives.

Guarantee by construction. A framework can guarantee by construction specific behavior if it enforce architectural constraints/styles. Dr-BIP allows the definition of configurations as parametric operators on components guaranteeing by construction specific properties. This is possible due to the runtime system model which captures, not only the system configuration, but also the possible configuration variants at design.

3.1.2 Conceptual Model

The conceptual model describes the abstract elements composing the Dr-BIP framework and the relation between them. In other words, it presents the broad picture of self-configuring systems modeled in Dr-BIP by introducing the framework's underlying concepts and new terminologies that will frame future discussions. The conceptual model is composed of two elements: the system model, and the adaptation engine. Figure 3.1 depicts the anatomy of the conceptual model. A description of each entity is presented next.

System Model. The system model is a representation of the running system. It is a form of runtime model, which provides a view on the running system and enables its adaptation. More information can be found on runtime models and

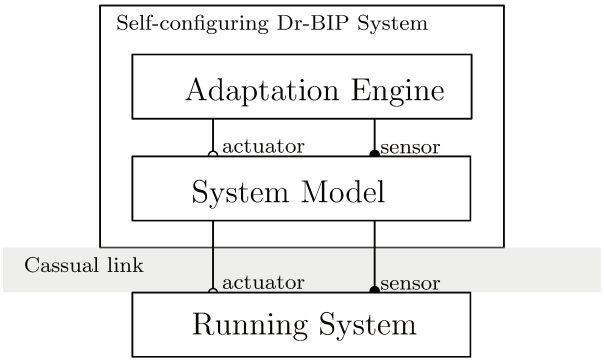


Figure 3.1: Conceptual model of Dr-BIP framework: dissecting the basic principles of a self-configuring system

their advantages, respectively, in sections 2.1, 2.2. In order to have a consistent view of the running system and to effectively modify the running system's configuration, the system model must be an accurate abstraction of the running system. To achieve this, Dr-BIP system model captures the running system at three different levels of abstraction. It captures both the system behavior and configuration during runtime. More over, it captures the system's possible configuration variants. The configuration encapsulates the system's current architecture describing its components and the interconnection between them. Configuration variants are expressed in terms of the configuration. A configuration variant is a constrained modification rule (reconfiguration rules) that uses operations for adding and removing components. Dr-BIP supports explicit addition and removal of components, but implicit addition and removal of connectors. The connection between components is defined by another set of rules (interaction rules) defined at design time. The main advantage of relying on an implicit addition and removal of connectors is the ability to guarantee by construction specific configuration topologies. To summarize configuration variant rules change the system's architecture.

Casual Link. The system model is casually connected to the running system in a bidirectional manner. The first direction assures that any change directed in the system model is also effective in the running system. The second direction assures that the system model is up to date with any change that happens at the running system. Therefore, it is important to maintain the conformance between the model and the running system in order to preserve consistency between them. To achieve the first direction, i.e. model to running system, mechanisms for comparing models, such a EMF [81], can be used to study the difference between the old and the new model. Then the difference can be used to generate scripts that modify the running system in the same way. To achieve the second direc-

tion, i.e. running system to model, reflective techniques can be used to introspect a running system and identify the exact changes made, which consequently can be used to update the system model. Maintaining the consistency in this way is inefficient and interferes with the performance of the system by introducing overhead. Dr-BIP avoids the hassle by capturing the system's architecture in its model and if we suppose that a mapping between components in the model and implementation modules is recorded. Then the mapping will enable changes specified in terms of the system model (addition/removal of component/connector) to be effective changes in the implementation and vice versa. In this manner, an up to date mapping maintains the correspondence between the system model and the implementation.

Adaptation Engine. In a nut shell, the adaptation engine monitors and controls the system model. It is responsible for continuously sensing the need for a reconfiguration, or a coordinated interaction between components. It does so by computing the set of enabled interactions and reconfigurations, deciding on an valid step (i.e. interaction or reconfiguration) and effecting the decision in the system model. The steps involved in the adaptation engine form an adaptation loop that is elaborated upon in the coming section.

The Dr-BIP conceptual model is similar to the general conceptual model proposed in 1.2.2 for self-adaptive systems. The managed system is represented in Dr-BIP by two elements, the system model and the running system. More over, the environment is implicitly captured in Dr-BIP 's system model by constrained configuration variants. For example, a server experiencing an enormous amount of tasks from users (environment) must adapt by delegating tasks to another server. In Dr-BIP this can be handled in the system model with a constrained reconfiguration rule, that migrates tasks to other servers when a server's load reaches a maximum value. Therefore, it is important to sufficiently study the environment of a system and how it behaves in order to capture it in the system model to account for and respond to any environmental change.

Adaptation Loop

The adaptation loop comprises the processes adopted by the adaptation engine to achieve self-configuration in Dr-BIP. Figure 3.2 illustrates the three main elements involved in self-configuration. The *model manager* maintains an up to date status of the system model. This up to date view is used to evaluate the current enabled interactions and reconfigurations. To compute the enabled interactions and reconfigurations, the *constraint evaluator* evaluates the constraints associated with each interaction and reconfiguration rule, which dictate its applicability (i.e. under which condition it applies). After evaluating the set of enabled steps, the

step executor is responsible for deciding on a step and directing the effect of actions associated with this step to the model manager which consequently affects the change in the system model through actuators.

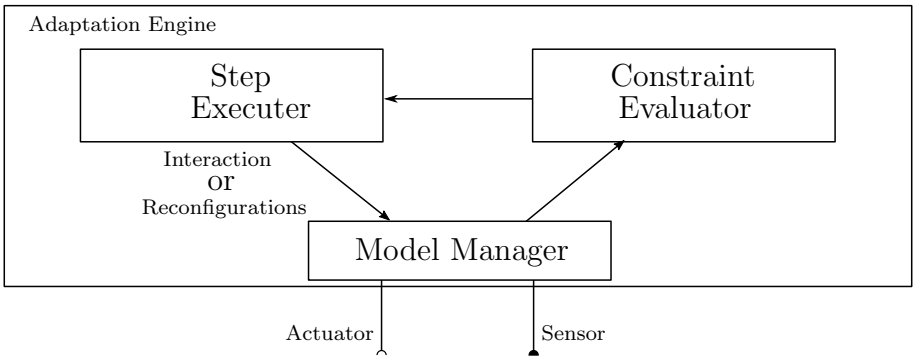


Figure 3.2: Adaptation loop describes the process adopted by the Dr-BIP adaptation engine to achieve reconfiguration

The Dr-BIP adaptation loop is similar to the MAPE-K loop described in section 1.2.2. The monitoring is handled by the model manager which keeps a consistent view of the current system model. Moreover, the analyzer is embedded in the constraint evaluator, however in Dr-BIP the analyzer is looking for a predefined patterns (enabled constraints) that trigger either a reconfiguration or an interaction. Each interaction/reconfiguration rule comes in a constraint, action pair. Therefore, the planning is inherently embedded in the interaction/reconfiguration rules. Once a step is chosen its set of associated actions are made effective in the system model. In summary, Dr-BIP relies on offline planning to achieve self-configuration. Finally the execution of action by the system model is made effective through the model manager and actuators.

3.2 Dr-BIP System Model

Dr-BIP framework utilizes runtime models to represent the running system. The use of runtime models to achieve self-adaptation have been discussed in details in section 2.1. In addition, an elaborate list of advantages for using runtime models to achieve self-configuration can be found in section 2.2. This section tackles the basic structure of a Dr-BIP system model and its composing elements. Henceforth, the words Dr-BIP model and system model will be used interchangeably used to signify the runtime abstraction model of the running system.

A Dr-BIP system model is an abstraction of the running system at three different levels of abstraction. Figure 3.3 summaries the three abstractions. The first

level captures the running system's functionality. For example in a client server system, the functionality of each server is captured by this layer. The Dr-BIP model captures system's behavior with the aid of automata extended with data and ports.

The second level abstracts the system configuration by encapsulating system's behavior in components and dictating their connections. For example, in a client server system, all clients and servers may be represented as components and an interaction may be used to to signal a connection between a client and server. The Dr-BIP model employs *interaction rules* to represent multiparty interaction between components.

The third level captures the variability in the system configuration. A system's *configuration* is determined by its components and their connections. A change in either the set of components or connections is said to be a *reconfiguration* that results in a new configuration of the system. The first type of reconfiguration that is responsible for changing the set of components in the system is handled explicitly by this layer. For example, in a client server system, a faulty server component may be removed and a new back up server may be introduced to cover up for the loss. Dr-BIP handles this type of reconfiguration by explicitly allowing the addition/removal of components through the definition of *reconfiguration rules*.

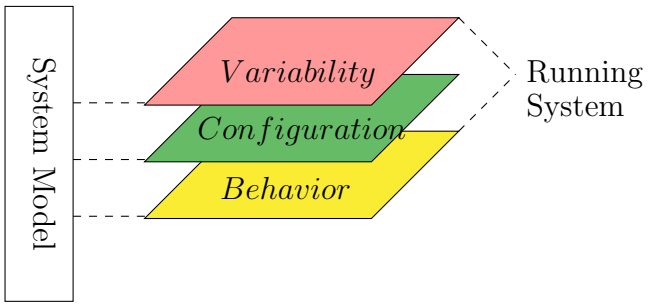


Figure 3.3: Dr-BIP system model abstracts the target system at three different levels of abstraction

The second type of reconfiguration which is responsible for modifying the connection/interaction between components is handled implicitly in Dr-BIP. Consider for example a client server system where only premium clients may be given access to servers with extra functionality. In other words, only premium client components can have a connection/interaction with special server components. Moreover, any new premium client must maintain such a connection. These constraints on connections are handled in Dr-BIP through parametric interaction rules, which belong to the second layer of abstraction, namely the configuration layer. An interaction rule can dictate that any client component that is of type premium must connect to server component of type x with special functionality. This rule

is applicable on all current premium clients in the system and will be applicable on any new premium clients introduced. In this way, the effect of reconfiguring the connection can only be seen in action once the component set in the system is changed by the introduction of a new premium client with the aid of reconfiguration rules. Therefore, as soon as a premium client component is introduced, its corresponding connection with other component servers is implicitly initiated according to interaction rules.

Therefore, reconfiguration is captured implicitly by the configuration layer (varying connection) and explicitly by the variability layer (varying components). Together these two layers capture the configuration space and possible configuration alternatives of the system.

3.2.1 Architecture Overview

To capture all three levels of abstractions discussed in section 3.2, the Dr-BIP framework introduces *architectural motif* as a key concept and an elementary unit of description for self-configuring systems. A motif encapsulates (i) behavior, as a set of components, (ii) interaction rules between components and (iii) reconfiguration rules about creating/deleting or moving components. A System is constructed as a superposition of several motifs, possibly sharing their components, and evolving altogether. The ability of components to move between motifs is a technique adopted by Dr-BIP to implicitly add/remove connectors. Connections between components in a motif is dictated by the motif's interaction rules. Dr-BIP employs a restricted approach that allows the ability to add/remove connectors implicitly, which in turn guarantees by construction certain configurations.

Figure 3.4 provides an overall view on the structure and evolution of a motif-based system (i.e. Dr-BIP model). The initial configuration on the left consists of six interacting components organized using three motifs. Motifs are indicated with dotted lines. The central motif contains components b_1 and b_2 connected in a ring. The upper motif contains components b_1, c_1, c_2, c_3, with b_1 being connected to all others. The lower motif contains connected components b_2, c_4.

The second system configuration in the middle shows a new configuration of the system after performing a reconfiguration. Component c_3 *moved* from the upper motif to the lower motif and Therefore, c_3 was implicitly disconnected from b_1 and connected to b_2 according to c_3's new motif's interaction rules. Note that the central motif is not impacted by the move.

The third system configuration on the right shows the system configuration after performing an additional reconfiguration. Two new components, b_3 and c_5, have been created. The central motif now contains an additional component, b_3, that is connected to b_1 and b_2, forming a larger ring. In addition, a new motif is housing the two newly created components b_3 and c_5.

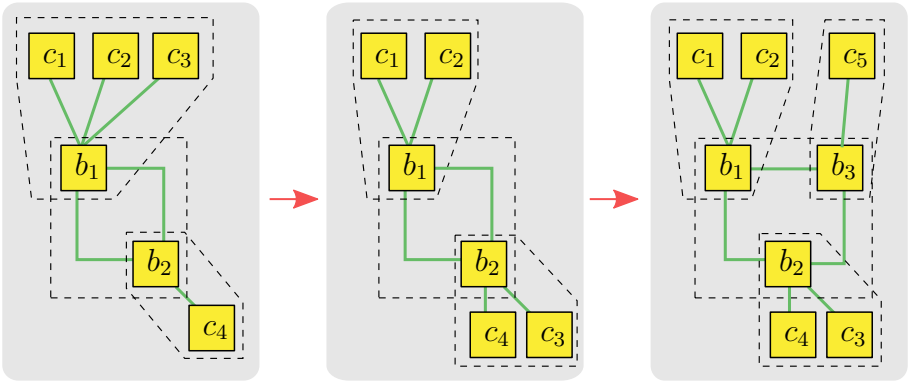


Figure 3.4: An example of reconfiguration in a motif-based system

The example above contains two types of motifs, a ring and star motifs. Motif types are defined by the types of hosted components along with parametric interactions and reconfiguration rules. Therefore, systems are described by superposing a number of such motifs on a set of components. In this manner, the overall system architecture captures specific configuration and functional properties by design.

Figure 3.6 depicts the definition of motifs in Dr-BIP. Motifs are structurally organized as the deployment of component instances on a logical map. *Maps* are arbitrary graph-like structures consisting of interconnected positions. *Deployments* relate component instances to positions on the map. Finally, the definition of the motif is completed by two sets of rules, defining respectively interactions and reconfiguration actions. *Interaction rules* defines a set of interactions between component instances. *Reconfiguration rules* defines reconfiguration actions to update the motif's components, map and/or deployment. The abstract syntax for both rules is presented in Figure 3.5.

interaction-rule ::=	*reconfiguration-rule* ::=
sync-*rule-name*(*formal-args*) ≡	**do-***rule-name*(*formal-args*) ≡
[**when** *rule-constraint*]	[**when** *rule-constraint*]
sync *interaction-ports*	**do** *reconfiguration-action*$^+$
[*interaction-guard* →	
interaction-action$^+$]	

Figure 3.5: Abstract syntax of interaction and reconfiguration rules

Both sets of rules are interpreted on the current motif structure. *Formal-args* denotes (sets of) component instances and defines the scope of the rule. *Rule-constraint* defines the conditions under which the rule is applicable. Constraints are essentially boolean combinations on deployment and map constraints built

from *formal-args*. An interaction rule also defines the set of interacting ports (*interaction-ports*), the interaction guard (*interaction-guard*) and the associated interaction actions (*interaction-action*). The guard and the action define respectively a triggering condition and an update of the data of components participating in the interaction. Finally, a reconfiguration rule defines reconfiguration actions (*reconfiguration-action*) to update the content of the motif. Such actions include creation/deletion of component instances, and change of their deployment on the map as well as change of the map itself, i.e. adding/removing map positions and their interconnection.

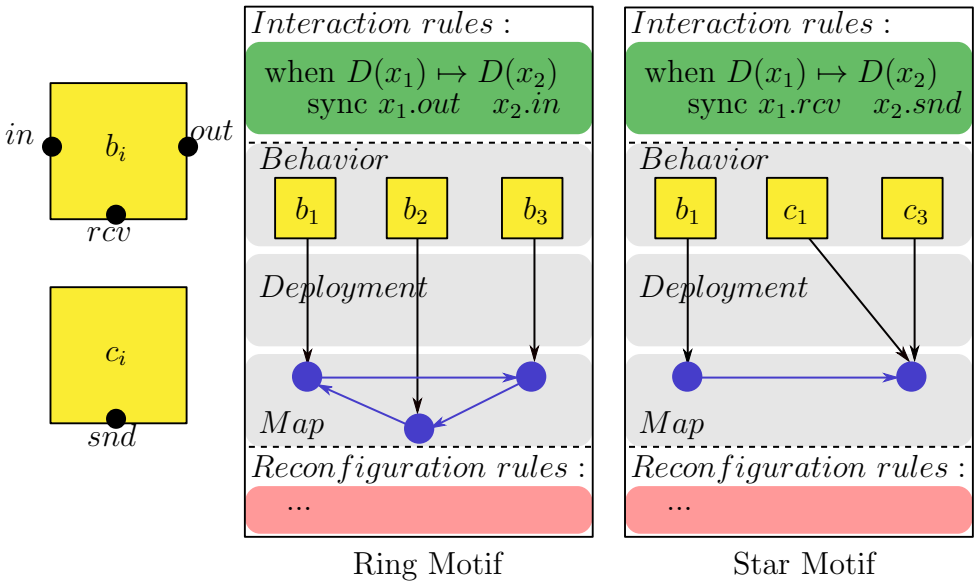


Figure 3.6: An example of a motif definition

The Ring motif illustrated on the left in Figure 3.6, defines the first type of motif used in the previous example. Three components b_1, b_2, b_3 are deployed into a three-position circular map. Given some deployment function D, the interaction rule reads as follows: for components x_1, x_2 deployed on adjacent nodes $D(x_1) \mapsto D(x_2)$ connect their ports $x_1.out$ and $x_2.in$. This rule defines three interactions between the b's components namely $b_1.out$ $b_3.in$, $b_3.out$ $b_2.in$, $b_2.out$ $b_1.in$ that correspond to the ring shown in Figure 3.4 on the right.

The "Star" motif illustrated on the right in Figure 3.6 defines the second type. Here, three components are deployed into a two-position map. The rule reads as follows: for components x_1, x_2 deployed on adjacent nodes $D(x_1) \mapsto D(x_2)$ connect their ports $x_1.rcv$ and $x_2.snd$. Given the current motif structure, the rule *defines* two interactions, namely $b_1.rcv$ $c_1.snd$ and $b_1.rcv$ $c_2.snd$, also illustrated in Figure 3.4 on the middle and right configurations.

Maps and deployments are chosen as a means for structuring components in a motif due to their simplicity. On one hand, maps and deployments are common concepts, easy to understand, manipulate and formalize. On the other hand, they adequately support the definition of arbitrarily complex sets of interactions over components by relating them to connectivity properties (neighborhood, reachability, etc). Moreover, maps and deployments are orthogonal to the behavior. Therefore they can be manipulated/updated independently and provide also a very convenient way to express various forms of reconfiguration.

Finally, the operational semantics of motif-based systems is defined in a compositional manner. Every motif defines its own set of interactions based on its local structure. This set of interactions and the involved components remain unchanged as long as the motif does not execute a reconfiguration action. Hence in absence of reconfigurations, the system keeps a fixed static architecture. The execution of interaction has no effect on the architecture. In contrast to interactions, reconfigurations rules are used to define explicit changes to the architecture, however, these changes have no impact on components. Therefore, all running components preserve their state although components may be created/deleted. This independence between execution steps is illustrated in Figure 3.7. A more concise definition of operational semantics for motifs and motif-based systems is provided in section 3.2.3.

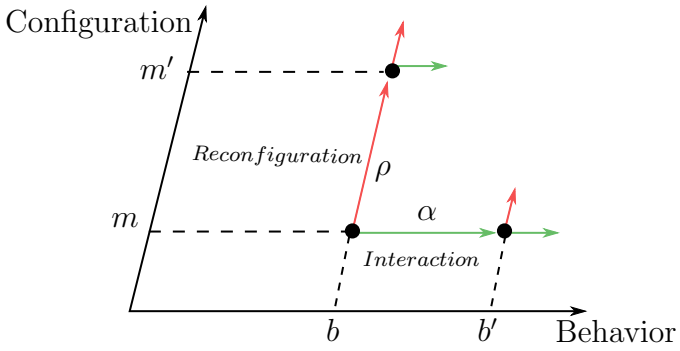


Figure 3.7: Reconfiguration vs Interaction Steps

In summary, Dr-BIP system model relies on motifs as a basic architecture unit of description. Dr-BIP systems are described by superposing a number of such motifs on a set of components. Therefore before elaborating on motif types and instances, the coming section first introduces component types and instances.

3.2.2 Components

Definition 1. *A component type B^t is an extended labeled transition system (L, P, V, T), where*

- *L is a finite set of control locations,*
- *P is a finite set of ports,*
- *V is a finite set of data variables,*
- *$T \subseteq L \times P \times \mathcal{G}(V) \times \mathcal{F}(V) \times L$ is a finite set of labeled transitions, where $\mathcal{G}(V)$ and $\mathcal{F}(V)$ are respectively Boolean guards and update functions defined over variables V.*

Every transition $\tau = (\ell, p, g, f, \ell') \in T$ is equivalently denoted as $\tau = \ell \xrightarrow{p\ g\ f} \ell' \in T$. For every port $p \in P$, we associate a subset of variables $V_p \subseteq V$ exported and available for interaction through p.

The set of states Q of a component type $B^t = (L, P, V, T)$ is defined by $Q = L \times \mathbf{V}$ where $\mathbf{V}$ is the set of all valuations defined on V. A valuation of a set of variables V is a function $\mathbf{v} : V \to \mathcal{D}$, where $\mathcal{D}$ is an underlying domain of data values. The semantics of a component type B^t is defined as the labeled transition system $[\![B^t]\!] = (Q, \Sigma, \to)$ where the set of labels $\Sigma = \{p(\mathbf{v}_p) \mid \mathbf{v}_p \in \mathbf{V}_p\}$ and transitions $\to \subseteq Q \times \Sigma \times Q$ are defined by the rule:

$$\frac{\tau = \ell \xrightarrow{p\ g\ f} \ell' \in T \qquad g(\mathbf{v}) \qquad \mathbf{v}''_p \in \mathbf{V}_p \qquad \mathbf{v}' = f(\mathbf{v}[\mathbf{v}''_p/V_p])}{B^t \ :\ (\ell, \mathbf{v}) \xrightarrow{p(\mathbf{v}''_p)} (\ell', \mathbf{v}')}$$

Therefore, $(\ell', \mathbf{v}')$ is a successor of $(\ell, \mathbf{v})$ labeled by $p(\mathbf{v}''_p)$ iff (1) $\tau = \ell \xrightarrow{p\ g\ f} \ell'$ is a transition of T, (2) the guard g holds on the current state valuation $\mathbf{v}$, (3) $\mathbf{v}''_p$ is a valuation of exported variables V_p and (4) $\mathbf{v}' = f(\mathbf{v}[\mathbf{v}''_p/V_p])$, which means that the next-state valuation $\mathbf{v}'$ is obtained by applying f on $\mathbf{v}$ previously updated according to $\mathbf{v}''_p$. Whenever a p-labeled successor exists in a state, we say that p is *enabled* in that state.

Example 1. *Figure 3.8 illustrates graphically a component type. It has three ports (in, out, rcv) attached with variables respectively u, v, and w. It has two control locations, namely (idle, busy) and three transitions labeled by its ports. For example, the transition labeled by in changes control location from idle to busy while performing the computation $v := u + w$.*

We consider a finite set of component types, fixed a priori. The component types facilitate the ability to describe parametric system coordination for arbitrary number of instances of component types. For example, systems with m Producers and n Consumers or Rings formed from n identical components etc.

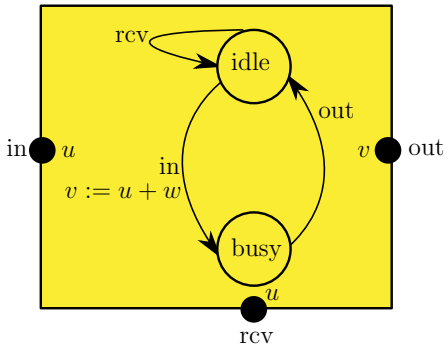


Figure 3.8: An example of a component type

Definition 2. *A component instance b is a couple (B^t, k) for some $k \in \mathbb{N}$. We denote respectively by ports(b), states(b), labels(b) the set of ports, states and labels associated with the instance b which are determined by its type.*

A system constructed from such component instances result in a BIP system [82, 83], which is later extended with motifs (Dr-BIP) to account for self-configuration. Motifs are detailed next in next section 3.2.3 and section 3.3 presents the semantics of a BIP component-based system.

3.2.3 Motifs

Motifs are dynamic structures composed of interacting components. The *structure* of a motif is expressed as a combination of three concepts namely, behavior, map and deployment. The behavior consists of a set of components. The map is an underlying logical structure (backbone) used to organize the interaction of components. The deployment provides the association between the components and the map. The components within a motif run in parallel and synchronize using multiparty interactions. The set of multiparty interactions is defined by interaction rules evaluated on the structure of the motif. Finally, the motif structure is also evolving. Any of the three constituents can be modified i.e., components can be added/removed to/from the motif, the map and/or the deployment can change. The motif evolution is expressed using reconfiguration rules, which evaluate and update the motif structure accordingly. Figure 3.9 graphically represents the underlying concepts of a motif and the relation between them. The motif's behavior has been introduced in the previous section 3.2.2 through the description of component types and instances. This section introduces formally all the remaining motif-related concepts.

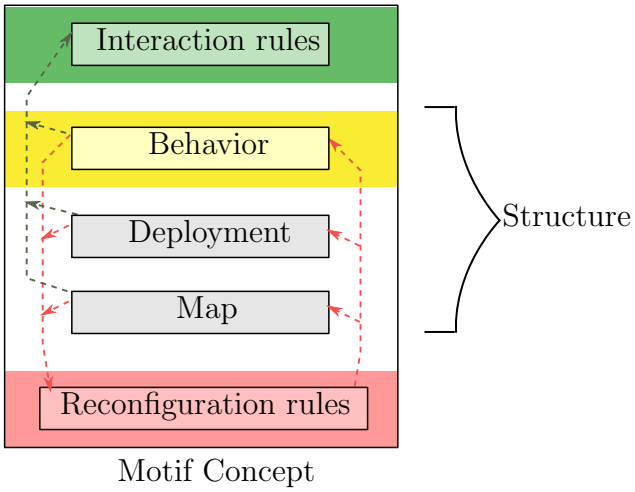


Figure 3.9: Overview of motifs structure and evolution rules

Maps and Deployments

Maps and deployments are abstract concepts used to organize the motifs. Maps denote arbitrary dynamic collections of inter-connected nodes (positions). They are defined as particular instances of generic map types.

Definition 3. *A map type H^t is characterized by $(N(H^t), \Omega(H^t), \mathcal{L}(H^t))$, where*

- *$N(H^t)$ is the underlying domain of nodes,*
- *$\Omega(H^t)$ is a set of primitives to update/access the map content, and*
- *$\mathcal{L}(H^t)$ is a logic to express constraints on the map content.*

We use maps as dynamically changing data structures (objects). The set of nodes of a map H is denoted by $dom(H)$. The dotted notation $H.op(\cdots)$ is used to denote the update and/or access to the map H according to $op \in \Omega(H^t)$. Moreover, $H \models \psi$ denotes that the constraint ψ is satisfied on map H for any $\psi \in \mathcal{L}(H^t)$.

Example 2. *In this example, we use a directed graph (V, E) as a map. Vertices V denote the positions and edges $E \subseteq V \times V$ express the connectivity between these positions. Such a map type (i) has the domain V, (ii) can be manipulated explicitly using primitives such as* addVertex, remVertex, addEdge, remEdge *and (iii) has predicate constraints such as edge constraints $\cdot \mapsto \cdot$, path constraints $\cdot \mapsto^* \cdot$), etc, with the usual meaning.*

Example 3. *Figure 3.10 to the right, illustrates a map type with a specific type of graph, namely, a cyclic graph, whose (i) vertices compose the domain and (ii)*

primitives include initialize, extend, remove *to respectively initialize, extend by one vertex and remove one vertex from it whilst keeping the cycling structure.*

Maps and behavior (component instances) are related through *deployments*. Deployments are partial mappings of a set B of component instances to the nodes of a map H, formally $D : B \to dom(H) \cup \{\perp\}$. Deployments are dynamic data structures defined as particular instances of a generic deployment types D^t. A D^t maintains a a set of primitives $\Omega(D^t)$ to update and/or access the deployment as well as a logic $\mathcal{L}(D^t)$ to express constraints on deployments.

Motif types

Definition 4. *A motif type M^t is a tuple $((\mathcal{B}, \mathcal{H}, \mathcal{D}), \mathcal{IR}, \mathcal{RR})$ where,*

- *the triple $(\mathcal{B}, \mathcal{H}, \mathcal{D})$ are motif meta-variables used to maintain respectively the set of component instances, the map and the deployment of component instances to the map,*
- *$\mathcal{IR}$ is a set of motif interaction rules of the form $(\mathcal{Z}, \Psi, P_I, G_I, F_I)$ where $\mathcal{Z}$ is a set of rule parameters, Ψ is a rule constraint, and (P_I, G_I, F_I) is the interaction specification, namely the set of ports of involved components, the guard and the data transfer, and*
- *$\mathcal{RR}$ is a set of motif reconfiguration rules of the form $(\mathcal{Z}, \Psi, G_R, \mathcal{Z}_L, A_R)$ where $\mathcal{Z}$ is a set of rule parameters, Ψ is a rule constraint, G_R is a reconfiguration guard, $\mathcal{Z}_L$ are local rule parameters, and A_R is a sequence of reconfiguration actions.*

The structure of a motif is defined by a consistent valuation of meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$ respectively as B, a set of components instances, H a map, and $D : B \to dom(H) \cup \{\perp\}$ a deployment. The structure can dynamically change as the meta-variables are being updated when reconfiguration rules are executed. We elaborate upon the meaning of these rules in subsequent sections.

Example 4. *Figure 3.10 presents a definition of the Ring motif type presented in section 3.2.1. The motif type contains one interaction rule denoted as* sync-inout *and three reconfiguration rules denoted respectively* do-init, do-insert *and* do-remove. *The motif structure is presented to the right of the figure. It is composed of a set of six component instances $B = \{b_i\}_{i=1,6}$, the map H defined as the cyclic graph of six nodes $\{n_i\}_{i=1,6}$, and the deployment $D = \{b_i \mapsto n_i\}_{i=1,6}$.*

sync-inout(x_1: C, x_2 : C) ≡ <u>when</u> D(x_1) ↦ D(x_2)
<u>sync</u> x_1.out x_2.in / true → x_2.u := x_1.v
do-init() ≡ <u>when</u> B = ∅
<u>do</u> x_1 := B.create(C, busy),
x_2 := B.create(C, idle), H.init(),
n_1 := H.extend(), D(x_1) := n_1
n_2 := H.extend(), D(x_2) := n_2
do-insert() ≡ <u>do</u> x := B.create(C, idle),
n := H.extend(), D(x) := n
do-remove(x : C) ≡ <u>when</u> |B| ≥ 3 ∧ x.idle
<u>do</u> n := D(x), B.delete(x), H.remove(n)

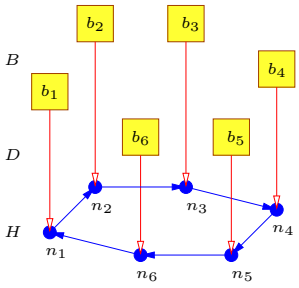


Figure 3.10: An example of a motif type

Interaction and Reconfiguration Rule constraints

A motif evolves according to its interaction and reconfiguration rules, which are respectively defined by the tuple ($\mathcal{Z}$, Ψ, P_I, G_I, F_I), and ($\mathcal{Z}$, Ψ, G_R, $\mathcal{Z}_L$, A_R). Before expanding on the syntax and meaning of each, we formally define the rule constraints Ψ in this section.

Rule constraints, Ψ, are boolean combinations of map constraints, deployment constraints and basic constraints built using parameters in $\mathcal{Z}$ and meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$. The rule parameters denoted by $\mathcal{Z}$ include typed symbols denoting (sets of) component instances or map nodes, which are interpreted as (subsets) elements of B or $dom(H)$ respectively. More precisely, Ψ is defined as follows:

$$\Psi ::= \psi^0 \mid \psi^{\mathcal{H}} \mid \psi^{\mathcal{D}} \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi$$

In the above definition, ψ^0 denotes a basic constraint built from equality and/or cardinality constraints on the parameters, $\psi^{\mathcal{H}}$ denotes a constraint on the map (conforming to the map logic $\mathcal{L}(H^t)$) and $\psi^{\mathcal{H}}$ denotes a constraint on the deployment (conforming to the deployment logic $\mathcal{L}(D^t)$).

Given a motif structure in terms of B, H, D, and an interpretation ζ of parameters, the satisfaction of a constraint $B, H, D, \zeta \models \Psi$ is defined recursively on the structure of Ψ as follows:

$$\begin{aligned}
B,H,D,\zeta \models \psi^0 &\quad \text{iff} \quad \zeta \cup [B/\mathcal{B}, H/\mathcal{H}, D/\mathcal{D}] \models \psi^0 \\
B,H,D,\zeta \models \psi^{\mathcal{H}} &\quad \text{iff} \quad H, \zeta \cup [B/\mathcal{B}, D/\mathcal{D}] \models \psi^{\mathcal{H}} \\
B,H,D,\zeta \models \psi^{\mathcal{D}} &\quad \text{iff} \quad D, \zeta \cup [B/\mathcal{B}, H/\mathcal{H}] \models \psi^{\mathcal{D}} \\
B,H,D,\zeta \models \Psi_1 \wedge \Psi_2 &\quad \text{iff} \quad B,H,D,\zeta \models \Psi_1 \text{ and } B,H,D,\zeta \models \Psi_2 \\
B,H,D,\zeta \models \neg\Psi &\quad \text{iff} \quad B,H,D,\zeta \not\models \Psi
\end{aligned}$$

In plain English, basic constraints are evaluated on the interpretation ζ extended with the current valuation of meta-variables $\mathcal{B}$, $\mathcal{H}$, and $\mathcal{D}$ in the usual way. Map constraints are evaluated on the map H and the interpretation ζ extended with the valuation of meta-variables $\mathcal{B}$, and $\mathcal{D}$ as defined by their underlying logic $\mathcal{L}(H^t)$. Deployment constrains are evaluated on the deployment D and the interpretation ζ extended with the valuation of meta-variables $\mathcal{B}$, and $\mathcal{H}$, as defined by their underlying logic $\mathcal{L}(D^t)$. Finally the evaluation of the conjunction of rule constraints is done in a compositional manner.

Interactions rules

Interaction rules are used to define multiparty interactions on the components instances within a motif. The syntax of the interaction specification part is as follows:

$$\begin{array}{rrcl} \text{ports:} & P_I & ::= & x.p \mid X.p \mid P_I\ P_I \\ \text{guard:} & G_I & ::= & \mathbf{true} \mid e_I \mid G_I \wedge G_I \mid \neg G_I \\ \text{action:} & F_I & ::= & \epsilon \mid x.v := e_I \mid X.v := e_I \mid a_I, a_I \\ \text{expression:} & e_I & ::= & x.v \mid X.v \mid op(e_I, \cdots, e_I) \end{array}$$

The symbols x, and X are rule parameters denoting respectively component instances or sets of component instances. Moreover, p is a component port, v is a component (exported) data variable and *op* is an operation on data values. A rule is syntactically well-formed iff all parameter names used in expressions (part of the guard or data transfer) are also used as part of the interacting port specification. In other words only data from components participating in the interaction can be used.

The set of multiparty interactions $\Gamma(r)$ corresponding to an interaction rule $r = (\mathcal{Z}, \Psi, P_I, G_I, F_I)$, for a given B, H and D of a motif is defined as:

$$\Gamma(r) = \left\{ (P_a, G_a, F_a) \;\middle|\; \begin{array}{l} B, H, D, \zeta \models \Psi \\ P_a = P_I(\zeta),\ G_a = G_I(\zeta),\ F_a = F_I(\zeta) \\ (P_a, G_a, F_a)\ \textit{well formed} \end{array} \right\}$$

The triple P_a, G_a, F_a is considered well formed iff it conforms to the definition of multiparty interactions, namely if P_a does not contain replicated or multiple ports of the same components, as well as if G_a and F_a use and update only variables exported on ports in P_a.

Example 5. *The ring motif type illustrated in Figure 3.10 has one interaction rule denoted by sync-inout. The rule connects the out port of a component x_1 to the in port of the component x_2 deployed next to it on the map. The resulting interactions are depicted in Figure 3.11.*

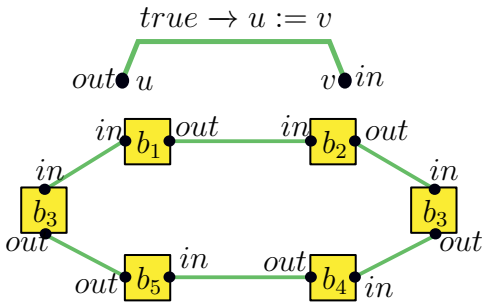


Figure 3.11: An example of a set of multiparty interactions in a motif

Reconfiguration rules

Reconfiguration rules are used to define actions that modify the structure of the motif. These actions essentially include creating/deleting component instances, updating the map and/or the deployment of component instances to the map. They are expressed as specific updates on the meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$. For enhanced expressiveness, reconfiguration rules might use additional local parameters (that is, the local context $\mathcal{Z}_L$) with arbitrary types (data, component instances, map nodes, etc). The local context is updated using standard assignments. The syntax of reconfiguration guards and actions is as follows:

$$\begin{array}{lrcl} \text{guard:} & G_R & ::= & G_I \\ \text{action:} & A_R & ::= & \epsilon \mid x := \mathcal{B}.create(B^t, q) \mid \mathcal{B}.delete(x) \mid \\ & & & \mathcal{H}.op_1(...) \mid \mathcal{D}.op_2(...) \mid z := e \mid A_R, A_R \end{array}$$

The symbol x denotes a rule parameter interpreted as component instance, z is an arbitrary local rule parameter and e is an arbitrary expression built on parameters and available operators. The intuitive meaning of reconfiguration actions is as follows. The action ϵ denotes an empty action with no effect. The action $x := \mathcal{B}.create(B^t, q)$ denotes the creation of a new component instance of type B^t with an initial state q. The newly created instance is x and is added to the set of components instances B. The parameter q denotes the initial state for the instance. The action $\mathcal{B}.delete(x)$ denotes the deletion of the component x from the motif, that is, the removal of the component instance x from the set B. The action $\mathcal{H}.op_1(...)$ denotes an update of the map according to an operator op_1 from $\Omega(H^t)$ and specific parameters. Similarly, the action $\mathcal{D}.op_2(...)$ denotes an update of the deployment according to an operator op_2 from $\Omega(D^t)$. Finally, the action $z := e$ denotes an update of a rule parameter according to the expression e.

Formally, the semantics $[\![A_R]\!]$ of a reconfiguration action A_R is defined as a function updating the motif structure (B, H, D), the set of component instances

$\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in \mathit{states}(b) \rangle$ and the parameter interpretation ζ, in the following manner:

$[\![\epsilon]\!](B, H, D, \mathbf{b}, \zeta) = (B, H, D, \mathbf{b}, \zeta)$
$[\![x := \mathcal{B}.\mathit{create}(B^t, q)]\!](B, H, D, \mathbf{b}, \zeta) = (B \cup \{b\}, H, D', \mathbf{b}', \zeta')$
where $b = (B^t, k), D' = D[b \mapsto \bot], \mathbf{b}' = \mathbf{b}[b \mapsto q], \zeta' = \zeta[x \mapsto b]$
$[\![\mathcal{B}.\mathit{delete}(x)]\!](B, H, D, \mathbf{b}, \zeta) = (B \setminus \{b\}, H, D_{|B\setminus\{b\}}, \mathbf{b}, \zeta)$ where $b = \zeta(x) \in B$
$[\![\mathcal{H}.op_1(...)]\!](B, H, D, \mathbf{b}, \zeta) = (B, H', D_{|H'}, \mathbf{b}, \zeta)$ where $H' = H.op_1(...)$
$[\![\mathcal{D}.op_2(...)]\!](B, H, D, \mathbf{b}, \zeta) = (B, H, D', \mathbf{b}, \zeta)$ where $D' = D.op_2(...)$
$[\![z := e]\!](B, H, D, \mathbf{b}, \zeta) = (B, H, D, \mathbf{b}, \zeta[z \mapsto e(\zeta \cup (B/\mathcal{B}, H/\mathcal{H}, D/\mathcal{D}))])$
$[\![A_{R1}, A_{R2}]\!](B, H, D, \mathbf{b}, \zeta) = ([\![A_{R2}]\!] \circ [\![A_{R1}]\!])(B, H, D, \mathbf{b}, \zeta)$

Example 6. *The ring motif type illustrated in Figure 3.10 contains three reconfiguration rules. The rule do-init initializes the motif with a ring of two components by first creating two components, initializing the map and extending it with two vertices's, and finally setting the deployment of the new components. The rule do-create adds a new component in the ring by creating a component of type C and adjusting the motif's map and deployment accordingly. The rule do-remove(x) removes an idle component x from the ring, only if the ring contains more than 3 components. The motif's map and deployment are adjusted accordingly.*

Operational semantics

A motif evolves by performing two categories of steps, namely interactions and reconfigurations. Interactions are defined by interaction rules and are executed by motif components. Reconfiguration are defined by reconfiguration rules.

Formally, the semantics of a motif type $M^t = ((\mathcal{B}, \mathcal{H}, \mathcal{D}), \mathcal{IR}, \mathcal{RR})$ is defined as the labeled transition system $[\![M^t]\!] = (Q, \Sigma, \rightarrow)$ where

- the states of set Q correspond to motif structure B, H, D consistently extended with configurations for all component instances $\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in \mathit{states}(b) \rangle$,
- the labels of Σ correspond to valid interactions α constructed on components and reconfiguration actions ρ,
- the transitions $\rightarrow = \underset{I}{\rightarrow} \cup \underset{R}{\rightarrow}$ correspond to execution of respectively multiparty interactions as defined by interaction rules ($\underset{I}{\rightarrow}$) and reconfiguration actions, as defined by reconfiguration rules ($\underset{R}{\rightarrow}$), formally

$$\text{(Mot-I)}\ \frac{\Gamma = \cup_{r\in\mathcal{IR}}\Gamma(r) \qquad \Gamma(B)\ :\ \mathbf{b} \xrightarrow{\alpha} \mathbf{b}'}{M^t\ :\ (B,H,D,\mathbf{b}) \xrightarrow[I]{\alpha} (B,H,D,\mathbf{b}')}$$

$$\text{(Mot-R)}\ \frac{\begin{array}{c}(\mathcal{Z},\Psi,G_R,\mathcal{Z}_L,A_R)\in\mathcal{RR} \qquad B,H,D,\zeta \models \Psi \\ G_R(\zeta)(\mathbf{b}) = \mathit{true} \qquad [\![A_R]\!](B,H,D,\mathbf{b},\zeta) = (B',H',D',\mathbf{b}',\zeta')\end{array}}{M^t\ :\ (B,H,D,\mathbf{b}) \xrightarrow[R]{\rho} (B',H',D',\mathbf{b}')}$$

In plain English, (Mot-I) says that the motif executes a multiparty interaction α and changes the configuration of components instances from $\mathbf{b}$ to $\mathbf{b}'$ iff (1) α belongs to the set of valid interactions Γ defined from the interaction rules and (2) a valid step labeled by α is indeed allowed between $\mathbf{b}$ and $\mathbf{b}'$ according to the component-based semantics discussed in 3.3. The rule (Mot-R) says that the motif executes a reconfiguration if (1) some reconfiguration rule is enabled at the current motif structure, and both constraint Ψ and guards G_R are satisfied for the given interpretation of parameter ζ and configurations of component instances $\mathbf{b}$ and (2) the current and next motif configuration are related according to the semantics of the action A_R. The dichotomy between interaction and reconfiguration steps ensures separation of concerns for execution within a motif as previously discussed in section 3.2.1 and illustrated in Figure 3.7.

3.2.4 Motif-based Systems

A Dr-BIP system is defined as a collection of motifs sharing a set of components. In such systems, every motif can evolve independently of the others, depending on its internal structure and associated interaction and reconfiguration rules. In addition, several motifs can also synchronize altogether and perform a joint reconfiguration over the system. Coordination between motifs is therefore possible either implicitly by means of shared components or explicitly by means of inter-motif reconfiguration rules. The inter-motif reconfiguration rules allow a joint reconfiguration of several motifs. In addition to the actions defined for reconfiguration rules, inter-motif reconfiguration rules introduce two additional types of actions, namely the creation and deletion of motif instances, and the exchange of component instances between motifs. Figure 3.12 illustrates the logical view of motif-based system. This section introduces formally inter-motif reconfiguration and defines the operational semantics of motif-based systems. We consider a finite set of motif types, fixed a priori and a motif instance m to be a tuple of the form (M^t, k) for some $k \in \mathbb{N}$.

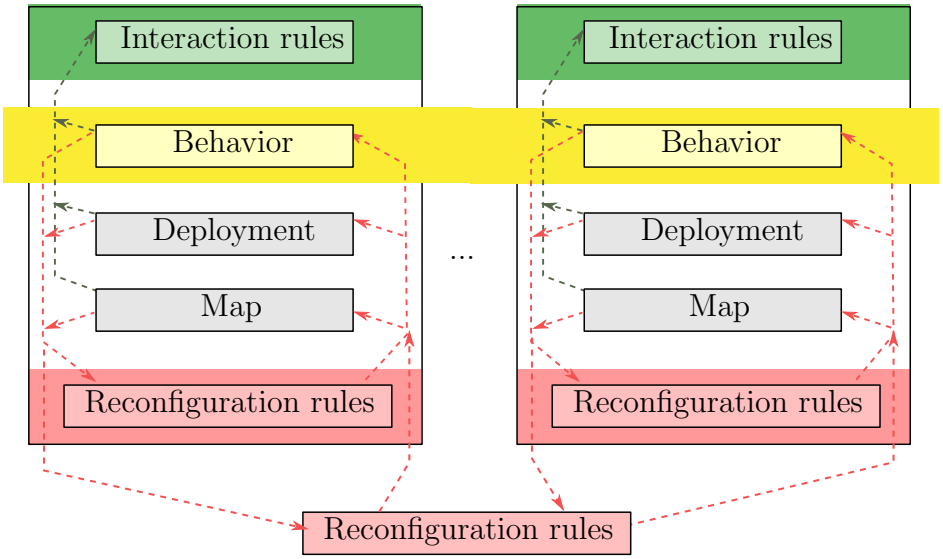


Figure 3.12: An Overview of motif-based systems

Inter-motif reconfiguration rules

Inter-motif reconfiguration rules, are similar to similar to local motif reconfiguration rules, and are defined as tuples of the form ($\mathcal{Z}^\star$,$\Psi^\star$, $G^\star$, $\mathcal{Z}^\star_L$, $A^\star_R$) . However, the set of rule parameter $\mathcal{Z}^\star$ might include additional symbols denoting motif instances (y). Moreover, the constraints Ψ^* are defined by the grammar:

$$\Psi^* ::= \Psi^{0*} \mid \langle y : \Psi \rangle \mid \Psi^*_1 \wedge \Psi^*_2 \mid \neg\Psi^*$$

In the above definition, Ψ^{0*} denotes basic equality and cardinality constraints expressed on parameter's interpretation, $\langle y : \Psi \rangle$ denotes a local constraint Ψ to be checked in the context of the motif instance y.

These constraints are evaluated on motif configuration extended with context parameters. Motif configurations are tuples $(M, \mathbf{m})$ where M is a set of motif instances and $\mathbf{m} = \langle m \mapsto (B, H, D) \mid m \in M \rangle$ provides the structure of these instances in terms of behavior, map and deployment. The constraints are evaluated as follows:

$$\begin{array}{rll} M, \mathbf{m}, \zeta \models \Psi^{0*} & \text{iff} & \zeta_{\mathbf{m}} \models \Psi^{0*} \\ M, \mathbf{m}, \zeta \models \langle y : \Psi \rangle & \text{iff} & B, H, D, \zeta_{\mathbf{m}} \models \Psi \text{ where } m \mapsto (B, H, D) \in \mathbf{m}, \zeta(y) = m \\ M, \mathbf{m}, \zeta \models \Psi^*_1 \wedge \Psi^*_2 & \text{iff} & M, \mathbf{m}, \zeta \models \Psi^*_1 \text{ and } M, \mathbf{m}, \zeta \models \Psi^*_2 \\ M, \mathbf{m}, \zeta \models \neg\Psi^* & \text{iff} & M, \mathbf{m}, \zeta \not\models \Psi^* \end{array}$$

In the above definition, $\zeta_{\mathbf{m}}$ denotes an extended context, including valuations for all meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$ accessed using parameters y of ζ. $\zeta_{\mathbf{m}}$ is defined as

follows:

$$\zeta_{\mathbf{m}} = \zeta \ \cup \ \langle y.\mathcal{B} \mapsto B, y.\mathcal{H} \mapsto H, y.\mathcal{D} \mapsto D \mid \zeta(y) = m, \ m \mapsto (B, H, D) \in \mathbf{m} \rangle$$

Inter-motif reconfiguration guards and actions are defined by:

$$\begin{array}{rrcl} \text{guard:} & G^{\star}_R & ::= & G_I \\ \text{action:} & A^{\star}_R & ::= & \epsilon \mid y := \mathcal{M}.create(M^t, (e_B, e_H, e_D)) \mid \mathcal{M}.delete(y) \mid \\ & & & y.\mathcal{B}.migrate(x) \mid \langle y : A_R \rangle \mid z := e \mid A^{\star}_R, A^{\star}_R \end{array}$$

In plain English, guards are the same as the guards of interaction rules. The action $y := \mathcal{M}.create(\ M^t, (e_B, e_H, e_D))$ denotes the creation of a new motif instance y of type M^t, with initial structure defined by the valuation of e_B, e_H, e_D. The action $\mathcal{M}.delete(y)$ denotes the deletion of the motif instance y, that is, its removal from the set of motif instances. The action $y.\mathcal{B}.migrate(x)$ denotes the insertion of an existing component instance x within the set of component instances of the motif y. Finally, the action $\langle y : A_R \rangle$ denotes any local reconfiguration action to be executed in the context of the motif instance y.

Formally, the semantics $[\![A^*_R]\!]$ of inter-motif reconfiguration actions is defined as a function updating motif configurations $(M, \mathbf{m})$, component configurations $(B, \mathbf{b})$ and context parameters (ζ), as follows:

$$\begin{array}{l} [\![y := \mathcal{M}.create(M^t, (e_B, e_H, e_D))]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M \cup \{m\}, \mathbf{m}', B, \mathbf{b}, \zeta') \\ \quad \text{where } m = (M^t, k), \ \mathbf{m}' = \mathbf{m} \cup \langle m \mapsto (e_B, e_H, e_D)(\zeta_{\mathbf{m}}) \rangle, \zeta' = \zeta[y \mapsto m] \end{array}$$

$$\begin{array}{l} [\![\mathcal{M}.delete(y)]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M \setminus \{m\}, \mathbf{m}_{|M \setminus \{m\}}, B, \mathbf{b}, \zeta) \\ \quad \text{where } m = \zeta(y) \in M \\ [\![y.\mathcal{B}.migrate(x)]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M, \mathbf{m}', B, \mathbf{b}, \zeta) \\ \quad \text{where } m = \zeta(y) \in M, m \mapsto (B_1, H, D) \in \mathbf{m}, \ \zeta(x) \mapsto b \in B, \\ \quad \mathbf{m}' = \mathbf{m}[m \mapsto (B_1 \cup \{b\}, H, D[b \mapsto \bot])] \\ [\![\langle y : A_R \rangle]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M, \mathbf{m}', B', \mathbf{b}', \zeta') \\ \quad \text{where } m = \zeta(y) \in M, m \mapsto (B_1, H, D) \in \mathbf{m}, \\ [\![A_R]\!](B_1, H, D, \mathbf{b}, \zeta) = (B'_1, H', D', \mathbf{b}', \zeta') \\ \quad \text{where } \mathbf{m}' = \mathbf{m}[m \mapsto (B'_1, H', D')], B' = B \cup B'_1 \\ [\![z := e]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M, \mathbf{m}, B, \mathbf{b}, \zeta[z \mapsto \zeta_{\mathbf{m}}(e)]) \\ [\![A^*_{R1}, A^*_{R2}]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = ([\![A^*_{R2}]\!] \circ [\![A^*_{R1}]\!])(M, \mathbf{m}, B, \mathbf{b}, \zeta) \end{array}$$

Example 7. *An example of a inter-motif reconfiguration rule that merges two ring motifs is presented below.*

do-merge(y_1, y_2 : Ring) $\equiv$
when $y_1.B \cap y_2.B = \emptyset$ *and* $|y_1.B| + |y_2.B| \leq 10$
do $B = y_1.B \cup y_2.B$, $D = y_1.D \cup y_2.D$, H = *merge-cycle($y_1.H$, $y_2.H$)*,
M.create(Ring, (B, H, D)), *M.delete(y_1)*, *M.delete(y_2)*

The rule allows the merging of two Ring motif instances y_1, y_2 into a single one, whenever their sets of component instances are disjoint and the cardinality of both motifs together does not exceed 10 component instances. The new motif is created by taking the union of component instances, the union of deployments and the merging of the two underlying cyclic maps. Finally, The original motifs y_1 and y_2 are deleted.

Operational semantics

A motif-based system $\mathcal{S}$ is defined as a tuple $((B_i^t)_i, (M_j^t)_j, \mathcal{RR}^*))$ consisting of a set of component types $(B_i^t)_i$, a set of motif types $(M_j^t)_j$ and a set of inter-motif reconfiguration rules $\mathcal{RR}^*$.

A motif-based system evolves either by executing local interactions and/or reconfiguration within any of the motifs, or by executing an inter-motif reconfiguration. Formally, the semantics of motif-based systems $\mathcal{S}$ is defined as the labeled transition system $[\![\mathcal{S}]\!] = (Q, \Sigma, \rightarrow)$ where:

- the set Q of system configuration contains tuples $(M, \mathbf{m}, B, \mathbf{b})$ where $M = \{m_1, m_2, ...\}$ is a set of motif instances, $\mathbf{m} = \langle m_j \mapsto (B_j, H_j, D_j) \mid m_j \in M,\ B_j \subseteq B \rangle$ are the motif configurations, B is the set of components instances, and $\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in \mathit{states}(b) \rangle$ are the component configurations,
- the set of labels Σ correspond to valid interactions α on component instances, local reconfiguration actions ρ and inter-motif reconfiguration actions ρ^*,
- the set of transitions $\rightarrow = \underset{I}{\rightarrow} \cup \underset{R}{\rightarrow} \cup \underset{R^*}{\longrightarrow}$ correspond to execution of respectively multiparty interactions as defined by interaction rules ($\underset{I}{\rightarrow}$), local reconfiguration as defined by local reconfiguration rules ($\underset{R}{\rightarrow}$) and global reconfiguration actions ($\underset{R^\star}{\longrightarrow}$), formally

$$\text{(M-I)}\ \frac{m_j \mapsto (B_j, H_j, D_j) \in \mathbf{m} \qquad M_j^t\ :\ (B_j, H_j, D_j, \mathbf{b}_j) \xrightarrow[I]{\alpha} (B_j, H_j, D_j, \mathbf{b}'_j) \qquad \mathbf{b}' = \mathbf{b}[B_j \mapsto \mathbf{b}'_j]}{\mathcal{S}\ :\ (M, \mathbf{m}, B, \mathbf{b}) \xrightarrow[I]{\alpha} (M, \mathbf{m}, B, \mathbf{b}')}$$

$$\text{(M-R1)}\ \frac{m_j \mapsto (B_j, H_j, D_j) \in \mathbf{m} \qquad M_j^t : (B_j, H_j, D_j, \mathbf{b}_j) \xrightarrow[R]{\rho} (B'_j, H'_j, D'_j, \mathbf{b}'_j) \qquad \mathbf{m}' = \mathbf{m}[(B'_j, H'_j, D'_j)/m_j] \qquad B' = B \cup B'_j \qquad \mathbf{b}' = \mathbf{b}[\mathbf{b}'_j/B'_j]}{\mathcal{S}\ :\ (M, \mathbf{m}, B, \mathbf{b}) \xrightarrow[R]{\rho} (M, \mathbf{m}', B', \mathbf{b}')}$$

$$\text{(M-R2)}\ \frac{(\mathcal{Z}^*, \Psi^*, G^*, \mathcal{Z}_L^*, A_R^*) \in \mathcal{RR}^* \qquad M, \mathbf{m}, \zeta \models \Psi^* \qquad G^*(\zeta)(\mathbf{b}) = \mathit{true} \quad [\![A_R^*]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M', \mathbf{m}', B', \mathbf{b}', \zeta')}{\mathcal{S} \ : \ (M, \mathbf{m}, B, \mathbf{b}) \xrightarrow[R^\star]{\rho^*} (M', \mathbf{m}', B', \mathbf{b}')}$$

In plain English, the rules (M-I) and (M-R1), respectively define the interaction and reconfiguration steps allowed within the motifs at the level of the system. The rule (M-R2) handles inter-motif reconfiguration. These transitions are allowed if (1) some inter-motif reconfiguration rule is enabled and (2) the current and next system configurations are related by the semantics of A_R^*.

3.3 Dr-BIP as an Extension of BIP

Dr-BIP is an extension to the BIP framework [82, 83] which facilitates the modeling of self-configuring component-bases systems. In BIP, systems are constructed from components, which are finite state automata, extended with data and ports. Communication between components is dictated by multiparty interactions with data transfer. BIP systems are static in the sense that components and interactions are fixed at design time and do not change during system execution. In other words, the system's configuration is known and fixed at design time. Transitioning from static systems to dynamic systems raises interesting questions. Is it possible to define a dynamic modeling language as an extension of a static modeling language? What is the relation between static and dynamic systems? In principle a dynamic system is more general, where by each intermediate configuration of a dynamic system actually corresponds to a static configuration. Therefore, any dynamic model can be converted to a static one, however such a conversion can lead to very complex systems. Dynamic system models result in more precise and concise models. Next, we briefly recall the key BIP concepts and their operational semantics. In addition we elaborate on existing BIP extensions for dynamic reconfiguration.

3.3.1 Component-based Systems

In BIP systems are composed of component instances and connector operators defining their interactions. An interaction is defined by a set of ports from various components. An interaction is *enabled* if there exists a set of enabled transitions labeled by its ports. The execution of an enabled interaction is followed by the completion of the involved transitions, which means the execution of their associated actions along with the change of the state of involved components to the target state. A formal definition is presented below.

Definition 5. *A system of components $\Gamma(B)$ is a tuple (B, Γ) where,*

- $B = \{b_1, ..., b_n\}$ *is a finite set of component instances, and*
- Γ *is a finite set of multiparty interactions. A multiparty* interaction *a is a triple (P_a, G_a, F_a), where*
  - $P_a \subseteq \bigcup_{i=1}^{n} ports(b_i)$ *is a set of ports, P_a must use at most one port of every component in B, that is, $|P_i \cap P_a| \leq 1$ for all $i \in \{1..n\}$. We denote $P_a = \{b_i.p_i\}_{i \in I}$, where $I \subseteq \{1..n\}$ contains the indices of the components involved in a and for all $i \in I, p_i \in ports(b_i)$,*
  - G_a *is a Boolean guard defined on the variables exported by ports in P_a (i.e., $\bigcup_{p \in P_a} V_p$), and*
  - F_a *is an update function defined on the variables exported by ports in P_a (i.e., $\bigcup_{p \in P_a} V_p$).*

The semantics of a system $S = \Gamma(B)$ is defined as the labeled transition system $[\![S]\!] = (Q, \Sigma, \rightarrow)$ where the set of states $Q = \langle b \mapsto q \mid b \in B,\ q \in states(b) \rangle$, the set of labels $\Sigma \subseteq \mathcal{P}(ports(B) \times \mathcal{P}(\mathbf{V}))$ contains the ports and sets of values exchanged on interactions and transitions $\rightarrow$ are defined by the rule:

$$\frac{\begin{array}{c} a = (\{b_i.p_i\}_{i \in I}, G_a, F_a) \in \Gamma \qquad G_a(\{\mathbf{v}_{p_i}\}_{i \in I}) \qquad \{\mathbf{v}''_{p_i}\}_{i \in I} = F_a(\{\mathbf{v}_{p_i}\}_{i \in I}) \\ \forall i \in I.\ \left(B_i^t \ :\ (\ell_i, \mathbf{v}_i) \xrightarrow{p_i(\mathbf{v}''_{p_i})} (\ell'_i, \mathbf{v}'_i) \right) \qquad \forall i \notin I.\ (\ell_i, \mathbf{v}_i) = (\ell'_i, \mathbf{v}'_i) \end{array}}{\begin{array}{c} \Gamma(B)\ :\ \langle b_1 \mapsto (\ell_1, \mathbf{v}_1), \ldots, b_n \mapsto (\ell_n, \mathbf{v}_n) \rangle \xrightarrow{\{b_i.p_i(\mathbf{v}''_{p_i})\}_{i \in I}} \\ \langle b_1 \mapsto (\ell'_1, \mathbf{v}'_1), \ldots, b_n \mapsto (\ell'_n, \mathbf{v}'_n) \rangle \end{array}}$$

In plain English, for each $i \in I$, $\mathbf{v}_{p_i}$ above denotes the valuation $\mathbf{v}_i$ restricted to variables of V_{p_i}. The rule expresses that S can execute an interaction $a \in \Gamma$ *enabled* in state $((\ell_1, \mathbf{v}_1), \ldots, (\ell_n, \mathbf{v}_n))$, iff (1) for each $p_i \in P_a$, the corresponding component instance b_i can execute a transition labeled by p_i, and (2) the guard G_a of the interaction holds on the current valuation $\mathbf{v}_{p_i}$ of exported variables on ports in a. Execution of a triggers first the update function F_a which modifies exported variables V_{p_i}. The new values obtained, encoded in the valuation $\mathbf{v}''_{p_i}$, are then used by the components' transitions. The states of components that do not participate in the interaction remain unchanged.

The semantics above provide the implementation basis of the BIP engine, which coordinates interactions between components. The engine is aware of the static set of interactions modeling the system. Therefore, it repeatedly executes the following three-step protocol: 1) each component sends the ports of its enabled transitions; 2) the engine computes the set of feasible interactions 3) the engine nondeterministically chooses an interaction and sends the names of ports involved for each involved components in the interaction. Figure 3.13 illustrates a static

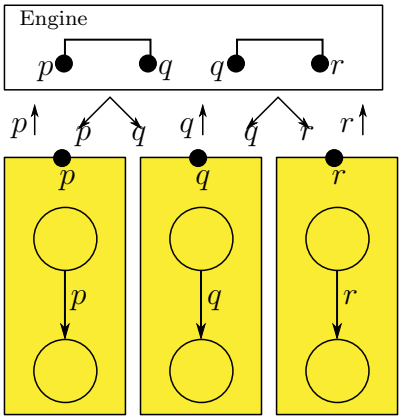


Figure 3.13: An example of a static configuration with BIP

system composed of three components, with three communication ports p,q,r, and defined by the interactions pq and qr.

3.3.2 Existing BIP Extensions for Dynamic Reconfiguration

Dy-BIP. Dy-BIP [84] has been introduced as a initial solution accounting for dynamic behavior in systems. While in BIP the set of interactions characterizing a system is static, in Dy-BIP it is dynamically changing. Dynamic systems are modeled in Dy-BIP as the composition of instances of component types. Dy-BIP relies on component types to represents a set of component instances having the same behavior and interface. This is particularly useful for systems that are built from multiple instances of components of different types.

To realize dynamic interactions, Dy-BIP introduces *interaction constraints* and *history variables*. An interaction constraints dictates how a component can interact with other component instances in the system. More precisely, it dictates which component ports are *required*, which component ports are optional (*accept*), and finally which component ports are excluded (*unique*), from an interaction. To concisely describe interaction constraints, Dy-BIP introduces an interaction constraint logic extended with first order logic and quantification over component instances. Formulas in this logic use port names as logical variables to characterize sets of interactions. A *feasible interaction* is any set of ports assigned true by a valuation which satisfies the formula.

Furthermore, Dy-BIP introduces history variables to keep track of executed interactions. History variables allow interactions to be enabled only if other interactions have been executed in the past. For example, consider a master slave system, where a master requests to work with two slaves, then the master must

interact with only the two accepting slaves. Therefore, history variables are used to parametrize interaction constraints. In other words, an interaction constraint may state the requirement of a certain port to be in the history variable for an interaction to be enabled.

In Dy-BIP, a component instance is an automaton extended with history variables and transitions on history variable. Each transition is labeled with a port, interaction constraint and history variable update. In other words, each component keeps track of the interactions it participated in within its history variable. Moreover, each port p of a component is now associated with an interaction constraints C_p, which describes the set of possible interaction involving that port. Therefore, the state of a component instance is dependent on its current control location and the valuation of its history variable (l,u). At each computational step component instances define their *state constraint*. A state constraint defines the set of possible interactions of a component at (l,u).

A Dy-BIP system consists of finitely many instances for each component type. The operational semantics of such a system are presented in [84]. The operational semantics are used to implement an engine which coordinates interaction between components by repeatedly executing the following three-steps: (1) each component instance sends its current state interaction constraint, (2) the engine builds the global system constraint by taking the conjunction of all state constraints and finds the set of maximally satisfying interactions (3) One of the maximal interaction is selected and executed by the involved components.

A Dy-BIP system can be seen an extension of BIP and a more general solution, since any static system modeled in BIP can be represented in Dy-BIP. For example, A BIP model with a static configuration constraint C, can be represented as a Dy-BIP model such that the constraint C_p associated with each port p is the set of interactions of C involving p. While Dy-BIP successfully introduces dynamic behavior to BIP systems, it still has major drawbacks. Dy-BIP assumes a finite set of component instances which are fixed a priori. Recall from section 1.2.2, that self-configuration of a system encompasses not only the dynamic change of connections between component, but also the dynamic change (creation/deletion) of component instances in the system. Dy-BIP tackles only the first type of self-configuration as it allows to model reconfiguration while varying connectivity between components. In Dr-BIP we encompass Dy-BIP and extend its expressiveness to all degrees of self-configuration.

Chapter

4

Dr-BIP by Examples

Contents

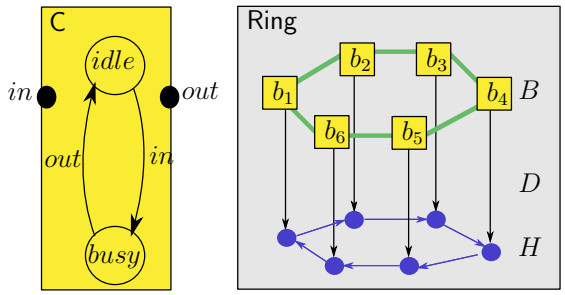
4.1 Self-configuring Token Ring System 56
4.2 Self-configuring Multicore Task System 59
4.3 Autonomous Highway Traffic System 63
4.4 Self-configuring Robot Colonies 66

This chapter presents four examples of self-configuring systems modeled in Dr-BIP. Each example is first introduced with an explanation of the intended target system's behavior and coordination. Next, motif structures along with component types are proposed to cater and compose a system model. Finally, interaction and reconfiguration rules are devised to achieve the target systems's behavior and coordination. Some examples are further accompanied with performance evaluations resulting from experiments with the Dr-BIP interpreter. The implementation details of Dr-BIP framework will be discussed in the coming chapter.

4.1 Self-configuring Token Ring System

Target System. A *token ring* consists of two or more identical components interconnected using uni-directional communication links according to a ring topol-

ogy. A number of tokens are circulating within the ring. A component is *busy* when it holds a token and *idle* otherwise. A component can do specific internal actions depending on its state, busy or idle. It can receive a token from the incoming link only if its idle. Moreover, it can send its token on the outgoing link only when its busy. A token ring is said to be *self-configuring* if idle components are allowed to leave the ring at any time leaving at least two components in the ring. In addition, new idle components are allowed to enter the ring at any time (as long as the maximal allowed ring size is not reached). A *token ring system* consists of one or more, pairwise disjoint, token rings. A token ring system is said to be *self-configuring* if every ring is self-configuring, and moreover, two rings are allowed to *merge* into a single one provided their overall size is not exceeding the maximal allowed ring size.



sync-ring-inout(x_1, x_2 : C) ≡ when D(x_1) ↦ D(x_2)
    sync x_1.out x_2.in

do-ring-insert() ≡ do x := B.create(C, idle), n := H.extend(), D(x) := n
do-ring-remove(x : C) ≡ when |B| ≥ 2 ∧ x.idle
    do n := D(x), B.delete(x), H.remove(n)
do-ring-merge(y_1, y_2 : Ring) ≡ when y_1.B ∩ y_2.B = ∅ ∧
            |y_1.B| + |y_2.B| ≤ 10
    do B = y_1.B ∪ y_2.B, D = y_1.D ∪ y_2.D, H = merge-cycle(y_1.H, y_2.H),
        create(Ring, (B, H, D)), delete(y_1), delete(y_2)

Figure 4.1: Self-configuring token ring system

Component Type. We propose a component type C whose behvaior is depicted in 4.1. It has two ports labeled *in*, and *out*, two control locations labeled *idle*, and

busy and two transitions labeled by its ports. For example, the transition labeled by *in* changes control location from idle to busy.

Motif structure. We propose a Ring motif type whose structure is depicted in Figure 4.1. The behavior B of the motif is a set of component instances of type C. The map H is a ring of locations, i.e. an instance of a cyclic directed graph map type. The map type's primitives include init, extend, remove, merge-cycle to respectively initialize, extend by one new location, remove one location and merge two cyclic directed maps. Moreover, the map type's predicates include $\cdot \mapsto \cdot$ to denote the presence of an outwards edge going from one location to the other. The deployment D function assigns components to locations in a bijective manner.

Interaction rules. We define a single interaction rule *sync-ring-inout*$(x_1, x_2 :$ C), which connects the *out* port of a component x_1, of type C, to the *in* port of the component x_2, of type C, deployed next to it on the map. The syntax of the interaction rule can be found in Figure 4.1

Reconfiguration rules. The motif reconfiguration is defined by three rules whose syntax can be found in Figure 4.1. The rule *do-ring-insert* creates a new component of type C having an initial state *idle*. The new component x is inserted in the ring by extending the motif's map and updating its deployment. The rule *do-ring-remove*$(x :$ C) removes an idle component x of type C from the ring, provided that the ring contains more than 2 components. Finally, the inter-motif reconfiguration rule *do-ring-merge* merges two ring instances y_1, y_2 into a single ring, whenever their sets of component instances are disjoint and together do not exceed 10. The new ring motif is created by taking the union of component instances, the union of deployments and the merging of the two underlying cyclic maps. The original rings y_1 and y_2 are deleted.

Figure 4.2 illustrates the execution of a dynamic ring system initialized with 10 ring motifs, each having 2 component instances. At each step, either an interaction or a reconfiguration (either within a motif or an inter-motif reconfiguration) is randomly executed. We remark that the number of ring motif instances decreases along the execution as idle components are removed and rings are enabled to merge into a single ring. The number of component instances varies across the execution between 6 and 20 as the *do-ring-insert* and *do-ring-remove* reconfiguration rules are executed.

Figure 4.2 summarizes the execution of the self-configuring ring system for different initial configurations, where the x-axis indicates the number of rings in the initial configuration and the y-axis is indicated at the top of each plot. We evaluate the performance and track the system evolution while varying the number

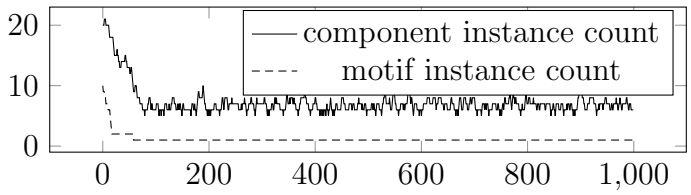


Figure 4.2: Dynamic ring system evolution across 1,000 steps

of initial rings from 10 to 100. Each configuration is simulated for 1000 random steps. As the system grows in size and the computation of enabled interactions and reconfigurations gets more complex, the execution time increases reaching a maximum of 14 seconds (first plot). The average ratio of the number of executed interactions vs reconfigurations along the run is around 0.45 (second plot). Finally, the minimum and maximum number of motif and component instances are depicted in the third and fourth plots.

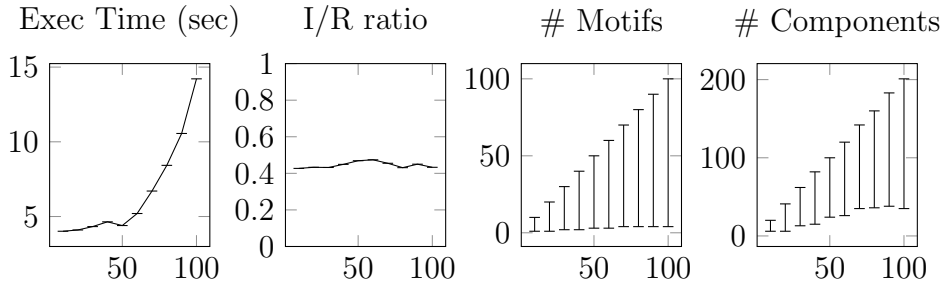
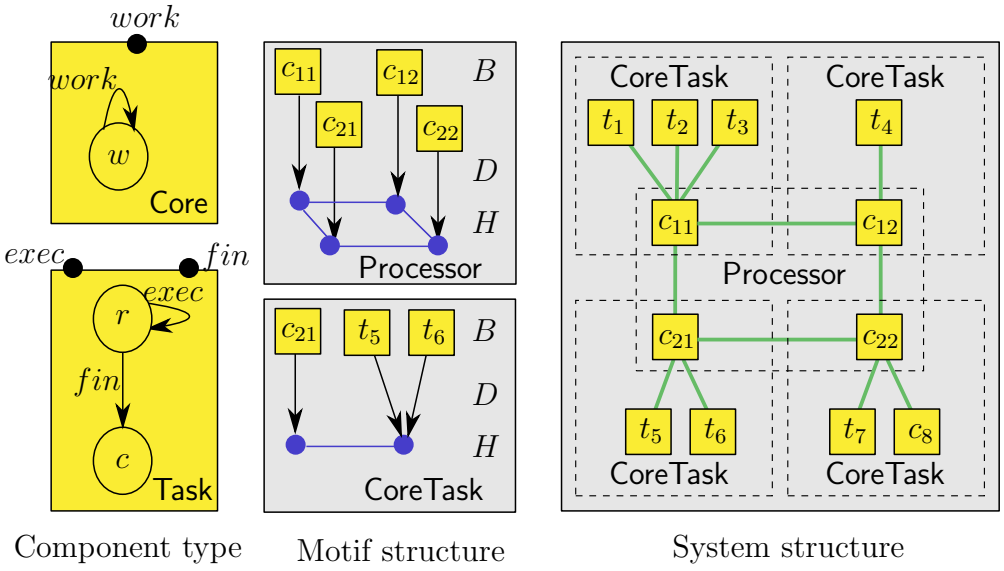


Figure 4.3: Self-configuring token ring system's measurements

4.2 Self-configuring Multicore Task System

Target System. This example considers task management for a multicore platform. Usually, tasks running on a multicore shall be evenly distributed amongst the cores so as to optimize the performance of the overall system. Task migration can be therefore seen as the result of a load balancing algorithm that aims at continuously improving the distribution of tasks amongst the cores. The multicore processor uses a scheduling algorithm to distribute the tasks over the cores. Therefore, a *multicore task system* consists of a fixed $n \times n$ grid of interconnected homogeneous cores c, each executing a finite number of tasks. A task can either be running or completed. Running tasks are executed on their associated cores and eventually get completed. The load of a core is defined as the number of its associated tasks including both running and completed tasks. A multicore task

system is *self-configuring* if the overall number of tasks and their allocation to cores may change over time. More specifically, new running tasks may enter the system at the core c_{11} and completed tasks may be withdrawn from the system at the core c_{nn}. Moreover, any task is allowed to migrate from its core to any of the neighboring cores (left, right, top or bottom) in the grid, provided the load of the receiving core is smaller than the load of the departing core minus some constant (K).



Component type Motif structure System structure

sync-coretask-exec(x_1 : Core, x_2 : Task) ≡ sync x_1.work x_2.exec
sync-coretask-fin(x : Task) ≡ sync x.fin

do-migrate(y_1, y_2 : CoreTask, y_3 : Processor, x_1, x_2 : Core, x_3 : Task) ≡
when $\langle\ y_1 : x_1 \in \mathsf{B}\ \rangle \wedge \langle\ y_2 : x_2 \in \mathsf{B}\ \rangle \wedge \langle\ y_3 : \mathsf{D}(x_1) \mapsto \mathsf{D}(x_2)\ \rangle \wedge$
$|y_1.\mathsf{B}| > |y_2.\mathsf{B}| + \mathsf{K} \wedge x_3 \in y_1.\mathsf{B}$
do y_2.migrate(x_3), y_1.delete(x_3)

Figure 4.4: Self-configuring multicore task system

Component type. We define two component types Core and Task whose behavior is depicted in Figure 4.4. The Core type has one control location labeled w, one port labeled *work*, and one cyclic transition labeled by its port. A core is continuously working to complete its associated task. The Task type has two control location r, and c, signifying respectively a running and a completed task.

It also has two ports labeled $exec$, and fin and two transitions labeled by its port.

Motif structure. We introduce two motif types, Processor and CoreTask, whose structure is depicted in Figure 4.4. The Processor motif type houses multiple interconnected core component instances (B). Their interconnecting topology reflects the architecture of the platform (e.g., a 2×2 grid in the Figure 4.4). The interconnection between cores is enforced with a grid-like map (H). The map type's predicate include $\cdot \mapsto \cdot$ to signify the presence of an edge between two location in the grid. The deployment (D) assigns the core component instances to locations in the map in a bijective manner. The CoreTask motif type is used to gather every core with its assigned tasks. In other words, its behvaior (B) is a set of component instances including a core and it's assigned tasks. The map (H) is composed of two locations connected with an edge. The deployment (D) assigns all task components to one location on the map, and the core component to the other location.

Interaction rules. We introduce two interaction rules to the CoreTask motif type. The first, *sync-coretask-exec*(x_1 : Core, x_2 : Task), synchronizes the execution of a task by its core. The second, sync-coretask-fin(x : Task), captures the completion of a task. The syntax of both interaction rules is presented in Figure 4.4.

Reconfiguration rules. The migration of a task from one core to another is modeled using an inter-motif reconfiguration rule, do-migrate(y_1, y_2 : CoreTask, y_3 : Processor, x_1, x_2 : Core, x_3 : Task), which involves three distinct motifs. The syntax of the rule is presented in Figure 4.4. The rule reads as follows in plain English, a task x_3 migrates from motif y_1 (of type CoreTask) to motif y_2 (of type CoreTask) if the core x_1 of y_1 is connected to the core x_2 of y_2 (according to the processor motif Processor) and if the number of tasks in y_1 exceeds the number of tasks in y_2 by constant K. To simplify notations in reconfiguration rules, we rely hence forth on sandwiching constraint/guard/action with angle brackets to specify the scope. For example $\langle y_1 : x_1 \in \mathsf{B} \rangle$ is a constraint stating that x_1 is a component instance in motif y_1. As the addition and removal of component instances to a motif has been illustrated in the previous example, we omit here for simplicity the reconfiguration rules which add tasks and remove tasks from the the top and rear core.

Figure 4.5 illustrates the execution of the self-configuring multicore task system with 3×3 cores by presenting the task load of each core across 3000 steps. Each core is initialized with a random load between 1 and 20. Moreover, The constant K is set to 3, hence tasks are allowed to migrate to neighboring cores (left, right,

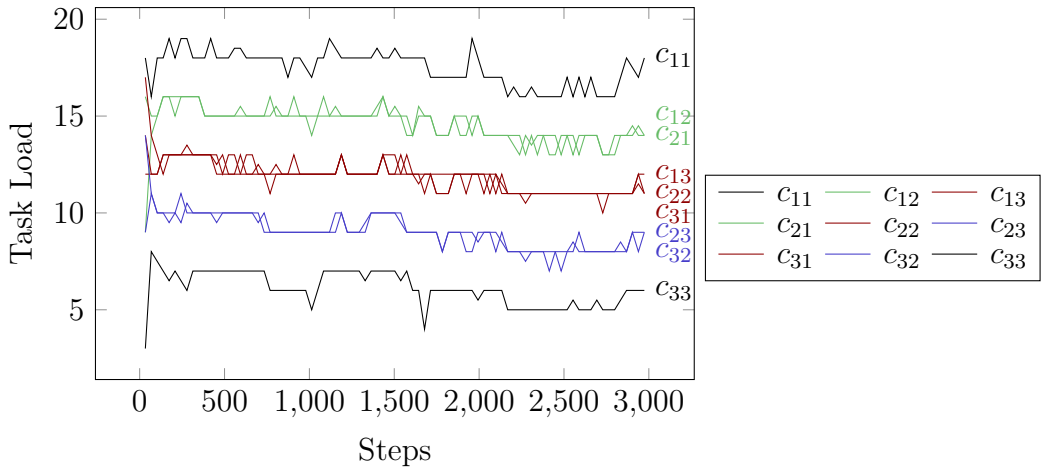


Figure 4.5: Task load across 3000 steps

top or bottom) that differ in task load by at least 3 tasks. The cores c_{11}, and c_{33} are used to respectively create new tasks and withdraw completed tasks. Note that the legend relates the core position to a color. The cores c_{11}, and c_{33} retain the maximum and minimum load. As tasks migrate, the task load of cores converges and balances along the execution resulting in at most a difference of 3 tasks between neighboring cores. As an example of task migration, take the core c_{21} whose task load is initially 6, and eventually reaches 14 as the execution continues. As expected the cores (c_{21}, and c_{12}) closest to c_{11} maintain a high task load and as we move away from c_{11} the core's task load gradually decreases. This highlights the task migration process cascading from the top left core to the bottom right core.

Figure 4.6 summarizes the execution of the self-configuring multicore task system for different initial configurations, where the x-axis indicates the number of motifs in the initial configuration (i.e. $n^2 + 1$ for $n = 2, 3, 4, 5, 6$) and the y-axis is indicated at the top of each plot. We vary the number of cores in the processor from 4 to 36 cores. Each core is initialized with a random load as discussed above. The system's initial size varies between 46 and 482 component instances as depicted in the figure. Each configuration is simulated for 1000 random steps. As the number of cores increases in size the execution time increases reaching a maximum of 7.3 seconds. The motif instance count remains constant across each configuration, however the component instance count varies as tasks are being created and deleted once completed. Also note that the average ratio of executed interactions vs reconfigurations is 0.7, since the task loads of cores eventually converges, leading to a similar value across cores and therefore resulting in less task migrations (i.e. reconfigurations).

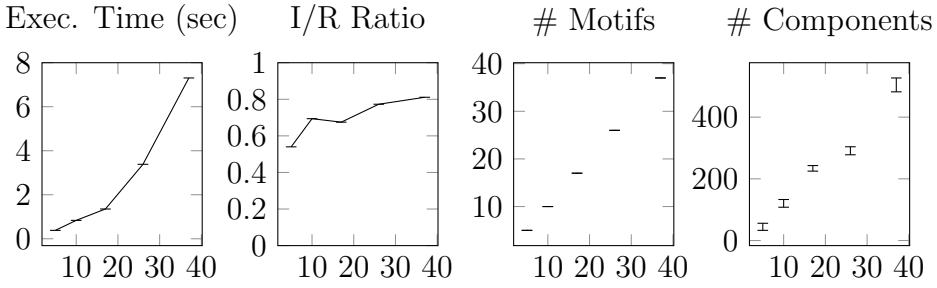
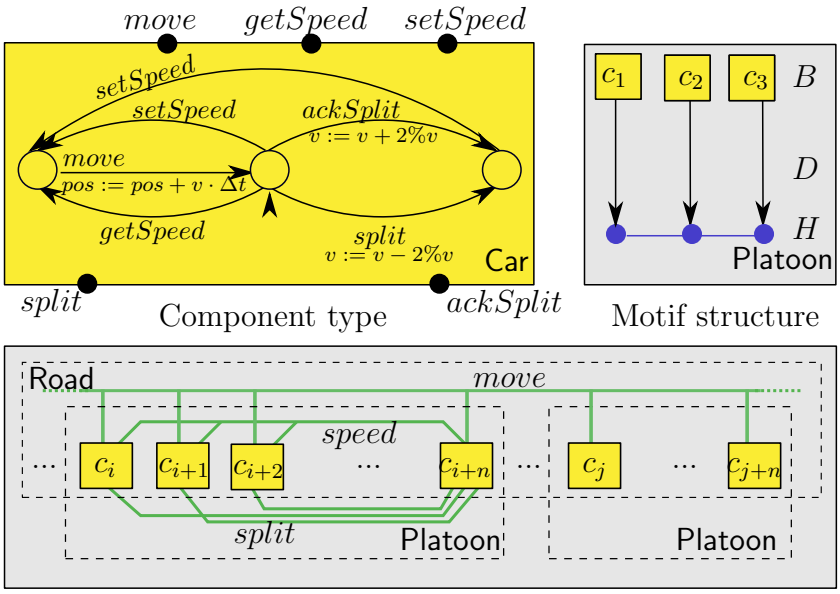


Figure 4.6: Self-configuring multicore task system's measurements

4.3 Autonomous Highway Traffic System

Target system. This example is inspired from autonomous traffic systems for automated highways [85]. The system consists of a single-lane one-way road where an arbitrary number of autonomous homogeneous self-driving cars are moving in the same direction, at different cruising speeds. Cars are organized into platoons, i.e. groups of cars cruising at the same speed and closely following a leader car. Platoons may *self-configure* by merging or splitting. A merge takes place if two platoons are close enough, i.e. the distance between the tail car of the first platoon and the leader car of the second is smaller than some constant K. After the merge, the speed of the new platoon is set to the speed of the first platoon. A platoon may split when an arbitrary car requests to leave the platoon e.g., in order to perform some specific maneuver. After the split, the leading platoon will increase its speed by 2% whereas the tail platoon will reduce its speed by 2%.

Component type. We propose a component type Car whose behavior is presented in 4.7. The Car type has three control location, each location signals the status of the car. For example, in the first state the car is in the moving state. In the second state, the car can update its speed according to its leader. Finally, the last state signals that the car has requested for a split. The initial control location is marked with an arrow head. Each Car maintains its position *pos* and speed v. Moreover, the Car has five ports *move, getSpeed, SetSpeed, split, ackSplit*, and 7 transitions labeled by its ports. The position *pos* is updated on the *move* transition. The transitions *setSpeed* and *ackSplit* are used by leader cars only to respectively define the platoon speed (i.e. update v) and acknowledge a request for a platoon split. Similarly, transitions *getSpeed* and *split* are used by follower cars only to respectively synchronize on the leader speed and initiate a platoon split.



sync-road-move(X : Car) ≡ when X=B sync X.move
sync-platoon-speed(x : Car, X : Car) ≡ when X=B \ x ∧ D(x) = H.head
    sync x.setSpeed X.getSpeed do $X.v = x.v$
sync-platoon-split(x_1, x_2 : Car) ≡ when D(x_1) = H.head ∧ $x_1 \neq x_2$
    sync x_1.ack_split x_2.split

do-platoon-merge(y_1, y_2 : Platoon, x_1, x_2 : Car) ≡
    when $\langle y_1$: D(x_1) = H.tail$\rangle$ ∧ $\langle y_2$: D(x_2) = H.head$\rangle$ ∧
        $|x_1$.pos $- x_2$.pos$| < K$
    do B := y_1.B ∪ y_2.B, H := append(y_2.H, y_1.H), D := y_1.D ∪ y_2.D,
        create(P, (B, H, D)), delete(y_1), delete(y_2)
do-platoon-split(y : Platoon, x : Car) ≡
    do $\langle y$: H_1 := H.sublist(0, D(x)), B_1 := $D^{-1}(H_1)$,
        D_1 := D.restrict(H_1), D_2 := D.restrict(H_2),
        H_2 := H.sublist(D(x), H.length), B_2 := $D^{-1}(H_2)$ $\rangle$,
        create(P, (B_1, H_1, D_1)), create(P, (B_2, H_2, D_2)), delete(y)

Figure 4.7: Automated Highway Traffic System

Motif Structure. We propose two motif types Road, and Platoon. A Road motif type contains all cars (B) and has no specific structure. Its map (H) is composed of a single node and its deployment (D) maps all cars to the same node. Due to its simplicity the motif structure of a Road is omitted from Figure 4.7. The Platoon motif type contains a bunch of cars forming a platoon (B). Its map is of a linear chain graph type (H) as shown in the motif structure in Figure 4.7. Moreover, the map type's primitives include head, tail, append, sublist and length. The head, and tail point to the leader and tail node, namely the beginning and end of the chain graph. the primitive append links two chain maps by appending them one after the other.The length returns the length of the chain. Finally, the sublist creates a sublist from a given chain map. The deployment (D) assigns each car in a platoon to a position on the map in a bijective manner. The deployment's primitive include restrict, which restricts a deployment by keeping only the mappings of components in a given map and removes the rest. Note that while $D(x)$ returns the node to which x is mapped, the inverse function $D^{-}1(H)$ returns the components that are deployed in a map H. This will deem useful when creating a sub platoon when a split is performed.

Interaction rules. The Road motif defines a single interaction by the rule *sync-road-move*, which synchronizes the move ports of all cars and therefore performing a joint update of their positions according to their speed. The Platoon motif defines interactions by two rules *sync-platoon-speed* and *sync-platoon-split*. The first rule synchronizes the speed of the leading car with the speed of all follower cars by setting the speed of all cars to the leader's speed. The second rule allows any non-leader car to initiate a split maneuver and become a leader in a newly created platoon. The syntax of these rules can be found in the Figure 4.7.

Reconfiguration rules. We define one reconfiguration rule *do-platoon-split* and one inter-motif reconfiguration rule *do-platoon-merge* to respectively split and merge platoons. The syntax of each rule can be found in 4.7. The *do-platoon-split* defines the structure of two motifs B_1, H_1, D_1 and B_2, H_2, D_2 which respectively, contain the cars, map and deployment of the sub-platoons. The motif structures are then used to create two newly platoons, and finally the existing platoon is deleted. The *do-platoon-merge* also constructs a motif structure composed of the union of the two platoon's behaviors B and deployments D. Moreover the platoon's maps are appended together (H). Finally, a new motif Platoon is created with reference to the created motif structure.

Figure 4.8 illustrates the evolution of the system involving 200 cars along 2000 sampled steps. Each line describes a configuration of the system. We show 13 sampled nonconsecutive configurations. A thin black rectangle represents a pla-

toon. Its length is proportional to the number of cars contained. Its position in the line corresponds to its position on the road. For reference, we show the evolution of a particular car by highlighting it in yellow. Initially, all the cars belong to the same platoon. As the system evolves the initial platoon splits into several platoons, which then keep splitting/merging back, etc.

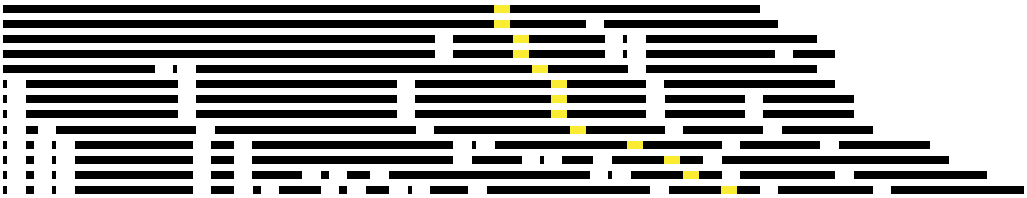


Figure 4.8: Automated highway traffic evolution along 13 sampled steps

Figure 4.9 summarizes the execution of several initial configurations, where the x-axis indicates the number of cars in the initial configuration and the y-axis is indicated at the top of each plot. We evaluate the performance and track the system evolution while varying the number of cars in the initial platoon from 200 to 600 cars. Each configuration is simulated for 3000 random steps. Notice that the component instance count remains constant across each configuration as cars only rearrange within different platoons. However the motif instance count varies as platoons merge/split. Finally, execution time increases reaching a maximum of 5 minutes and the average ratio of executed interactions vs reconfigurations is 0.77.

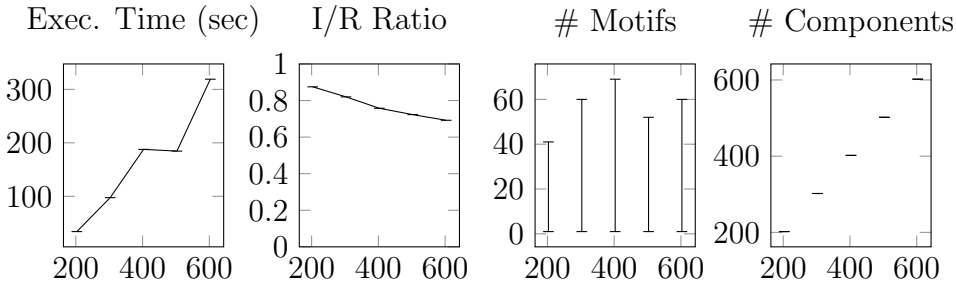


Figure 4.9: Automated highway traffic system's measurements

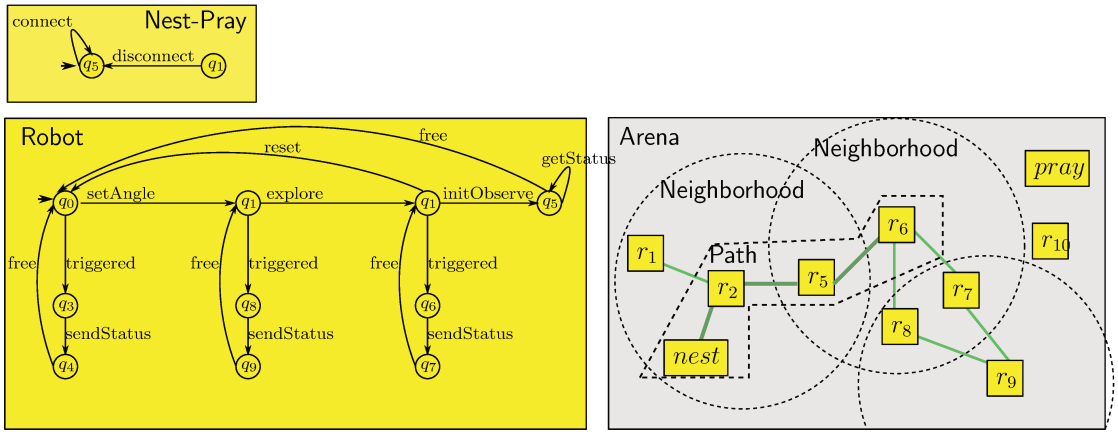
4.4 Self-configuring Robot Colonies

Target system. This exercise is inspired by swarm robotics [86]. A number of identical robots are randomly deployed on a field and have a mission to locate an

object (the prey) and moves it near another object (the nest). The robots know neither the position of the nest nor the position of the prey. They have limited communication and sensing capabilities, i.e. they can display a status (by turning on/off some colored leds) and can observe each other as long as they are physically close in the field. We consider hereafter the swarm algorithm proposed in [86]. In a first phase, the robots self-organize into an exploration path starting at the nest. The first robot detecting the nest initiates the path, i.e. stops moving and displays a specific (on-path) status. Any robot that detects (robots on) the path, begins moving along the path towards its tail, explores a bit further its neighborhood and gets connected as well (i.e. becomes the new tail, stops moving and displays the on-path status). Two cases may occur, either no new robot gets connected to the path within some delay, hence the tail robot disconnects and moves randomly (away from the path), or the tail robot detects the prey and the second phase starts. The path stays in place while additional robots converge near the prey. When enough robots have converged, they start pushing the prey along the path towards the nest. The path gets consumed, and the system will stop when the prey gets close enough to the nest.

Component type. We propose three component types Nest, Pray, and Robot whose behavior is presented in Figure 4.10. The Nest and Pray have similar behaviors. They maintain their position in the arena, two control location, two ports (*connect*, and *disconnect*), and two transition on their ports. Once a robot is close enough to a nest/pray, it connects to it through an interaction with the *connect* port. The robot disconnects from the nest if no new robots connect to it through an interaction with the *disconnect* port. Each Robot component keeps track of its position, angle of movement, number of robots in its neighborhood, target position, a boolean to indicate whether it is on the path or not.

Motif structure. We propose three motif types Arena, Neighborhood and Path. The Arena motif contains all the robots, the nest and the prey component instances (B). Its map is composed of a single node (H), with no specific structure. The deployment maps all component instances to the singleton map location (D). Each robot component is contained within a Neighborhood motif, which represents its visibility range. In other words, the Neighborhood motif contains a robot along with the set of robot instances that are physically close to it (B). The Neighborhood's map is composed of two connected nodes. The deployment maps the center robot to the first location, and all neighboring robots are mapped to the second location. The map's primitives include center, neighbor. The center, and neighbor primitives points to, respectively, the node which the center or neighbor robot is deployed to. Moreover, the deployment is accompanied with a primitive remove, which removes



Component types System structure

sync-set-angle(X : Robot) ≡ when X=B sync X.setAngle
sync-explore(X : Robot) ≡ when X=B sync X.explore
sync-init-observe(x : Robot, X : Robot) ≡
    when X=B \ x ∧ D(x) = H.center sync x.initObserve X.triggered
sync-updateStatus(x : Robot, X : Robot) ≡
    when X=B \ x ∧ D(x) = H.center ∧ X.onPath = true
sync x.getStatus X.sendStatus
    do $x.targetPos = X.Pos$
sync-free(x : Robot, X : Robot) ≡ when X=B \ x ∧ D(x) = H.center
    sync x.free X.free
sync-reset(x : Robot) ≡ when D(x) = H.center sync x.reset

Figure 4.10: Self-organizing robot colonies

the deployment of a neighbor once it leaves the visibility range of the center robot. Finally, the Path motif contains all robots forming a path connecting the nest and the pray (B). Its map type is composed of a linear chain graph and its deployment maps the components in a bijective manner. This map type is accompanied with primitives textsftail, extend, and *remove*, to respectively get the tail node, extend the chain by one node, and remove a a node form the chain. Due to the similarity of these motif structures with motif structures from the previous examples we omit their illustration in the Figure 4.10.

Interaction rules. The Arena motif defines a global interaction *setAngle* which sets the angle/direction of movement of all robots. The angle is set either randomly

or to a specific value computed by the angle between two positions *pos* (position of robot) and *targetPos* (position of an observed robot on path). The *targetPosition* is updated when the robot observes a robot on the path in its neighborhood. The angle assignment is handled internally within the robot using internal actions on its transitions. The Arena defines another global interaction *explore* used to model the synchronous passage of time along with the exploration of the robots within the arena. When this interaction is triggered the robots update their positions according to the current position, and target angle. These actions are handled on internal transition of a robot type.The motif Neighborhood defines four interaction rules *initObserve*, *observer*, *free*, and *reset*. The *initobserve* defines a set of binary interactions which are used by a robot to initiate the observing stage, a stage by which a robot updates its information from its neighbors. The *updateStatus* defines a set of binary interactions which are used by a robot to collect all the available information from its neighbors. This interaction updates the robot's *targetPos*. Once the robot has observed all its neighborhood, it releases its neighborhoods in order to continue exploring, it does so through the *free* interaction rule. Moreover, if a robot does not wish to initiate an observation of its neighborhood it resets, using the *reset* interaction so that robots can continue computing their angle of direction and exploring the arena. Finally, the Path motif defines a set of binary next, and *prev* interactions which are used to communicate along the path.

Reconfiguration rules. Reconfiguration rules are used to redefine the content of the Neigborhood and Path motifs. The syntax of these rules is presented in Figure 4.11. As robots are moving in the arena, they continuously enter or leave the visibility range of other robots. We use two inter-motif reconfiguration rules to update the neighborhood information, namely *do-neighborhood-enter*, and *do-neighborhood-leave*. The rules above describe the reconfiguration allowing any robot x_2 to enter (resp. leave) the neighborhood y_1 of any different robot x_1 whenever the distance between x_1 and x_2 is smaller than R_{min} (resp. greater than R_{max}). The evolution of the Path is also described by two reconfiguration rules, namely *do-path-connect*, and *do-path-disconnect*. These rules capture respectively the connection of a robot to the tail of a path At any time, and its disconnection.

do-neighborhood-enter(y_1 : Neighborhood, y_2 : Arena, x_1, x_2: Robot) ≡
when $\langle y_1 : \mathrm{D}(x_1) = \mathrm{H.center} \wedge x_2 \notin \mathrm{B} \rangle$
$\wedge \langle y_2 : x_2 \in \mathrm{B} \rangle \wedge \mathrm{dist}(x_1, x_2) \leq R_{min}$
do y_1.migrate(x_2), $\langle y_1 : \mathrm{D}(x_2) := \mathrm{H.neighbor} \rangle$
do-neighborhood-leave(y_1 : Neighborhood, x_1, x_2: Robot) ≡
when $\langle y_1 : \mathrm{D}(x_1) = \mathrm{H.center} \wedge x_2 \in \mathrm{B} \rangle$
$\wedge\, x_1 \neq x_2 \wedge \mathrm{dist}(x_1, x_2) \geq R_{max}$
do $\langle y_1 : \mathrm{D.remove}(x_2), \mathrm{B.delete}(x_2) \rangle$
do-path-connect(y_1 : Path, y_2 : Neighborhood, x_1, x_2 : Robot) ≡
when $\langle y_1 : \mathrm{D}(x_1) = \mathrm{H.tail} \wedge x_2 \notin \mathrm{B} \rangle$
$\wedge \langle y_2 : \mathrm{D}(x_1) = \mathrm{H.center} \wedge x_2 \in \mathrm{B} \rangle$
do y_1.migrate(x_2), $\langle y_1 : n = \mathrm{H.extend}(), \mathrm{D}(x_2) := n \rangle$
do-path-disconnect(y_1 : Path, x_1 : Robot) ≡
when $\langle y_1 : \mathrm{D}(x_1) = \mathrm{H.tail} \rangle \wedge \langle y_1 : x_1.\mathit{timeout} = \mathrm{true} \rangle$
do $\langle y_1 : n := \mathrm{D}(x_1), \mathrm{B.delete}(x_1), \mathrm{H.remove}(n) \rangle$

Figure 4.11: Reconfiguration rules of a self-configuring robot system

Chapter

5

Implementation

Contents

5.1 Overview 71
5.2 Concrete Syntax 72
5.2.1 Lexical Structure 73
5.2.2 Grammar Highlights 75
5.2.3 An Example Using The Concrete Syntax 79
5.3 Parser 80
5.4 Interpreter 81
5.4.1 Parameters 82
5.5 Execution 82

5.1 Overview

The Dr-BIP framework employs a model-based approach, where by models are the primary artifacts of description. A model is typically expressed with a modeling language, which comprises an abstract syntax, concrete syntax and semantics. Chapter 3 presented the framework concepts along with the abstract

syntax and semantics of the modeling langauge. Chapter 4 illustrated how various self-configuring systems can be modeled in Dr-BIP whistling utilizing the abstract syntax. This chapter introduces our prototype implementation of the Dr-BIP framework which includes a concrete syntax to describe motif-based systems, a parser and an interpreter for the operational semantics.

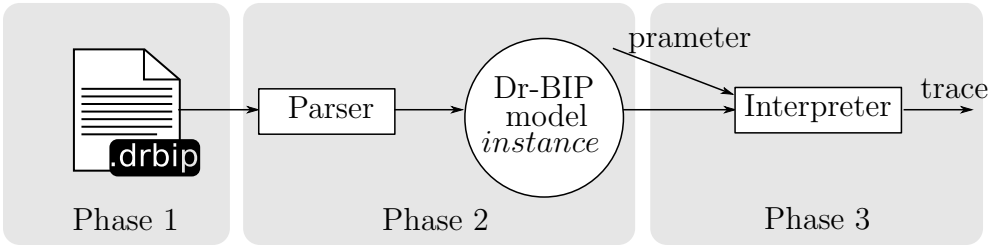


Figure 5.1: An overview of the prototype implementation

Figure 5.1 illustrates the prototype implementation across three phases. The first phase, introduces a concrete syntax which defines rules on how to textually describe a motif-based system. A system description/code that conforms to the concrete syntax is presented in the Figure as a file with ".drbip" extension. The second phase introduces a parser that scans the input file (i.e. system description code) and outputs an instance of a system model that corresponds to the input description. Finally, the third phase introduces an interpreter that uses the framework's operational semantics to interpret a given model instance. The interpreter requires certain parameters such as the number of steps to execute. A step could either be an enabled reconfiguration or interaction. Other parameters will be presented later on. These three phases are described in details respectively in section 5.2, 5.3, and 5.4. Both the parser and interpreter have been developed in JAVA and together form over 10,800 lines of code divided across 226 classes and 22 packages.

5.2 Concrete Syntax

This section takes the syntax proposed in chapter 3 from an abstract level to a more concrete level. A concrete syntax assigns textual notations or syntactical constructs to elements that are defined in the abstract syntax. This implies that our concrete syntax will dictate how to syntactically define motif-based systems, which includes the definition of component types/instances, motif types/instances, interaction rules, and reconfiguration rules etc. The smallest unit of a concrete syntax is a *lexeme*. A lexeme –also known as a token– is a sequence of characters. A Dr-BIP system description can be viewed as a stream of lexemes. Therefore,

the creation of a concrete syntax involves two steps, 1) specifying *what* are the lexemes of a language, and 2) *how* these lexemes can be grouped and arranged.

The following section (5.2.1) tackles the first step by introducing the lexical structure, which lists the lexemes of the concrete syntax. Section 5.2.2 tackles the second step by introducing a grammar, which is a set of rules that defines how lexemes can be arranged. Finally, section 5.2.3 presents an example of a Dr-BIP system defined using the concrete syntax.

5.2.1 Lexical Structure

The lexical units can be broadly categorized into integers, identifiers, special symbols, keywords, and white space. Each category defines a set of Lexemes, which are summarized in Table 5.1.

Integers, Identifiers and Special notation. *Integers* (*INT*) are strings with digits between [0-9]. *Identifiers* (*ID*) are strings consisting of a letter followed by letters, or digits. For example, identifiers include port names, component and motif type names, component data names, component state names etc. The special symbols (*SPECIAL-SYMBOLS*) include brackets, mathematical operator, boolean operators and others given in Table 5.1.

keywords. *Keywords* (*KEY-WORDS*) are reserved identifiers. Dr-BIP concrete syntax includes keywords such as **system**, **component**, **port**, **motif**, **for**, **each**, **interaction**, **in**, **initial**, **to**, **from**, **on**, **where**, **true**, **false**, **int**, **boolean** and others defined in Table 5.1. Note that keywords are case sensitive. For example, System, and SySTem are not treated as keywords.

white space. *White space* (*WS*) consists of any sequence of the characters composed of blank, \n, \r carriage return, or \t.

comments. We extend the lexical structure with two forms of comments to aid system designers with adding textual content that will not be parsed and interpreted i.e. will not be considered part of the Dr-BIP system description. The first defines a single line comment and is composed of any characters between two forward slashes (//) and the next newline or carriage return. The second defines a multi-line comment (i.e. spans multiple lines) and is composed of any characters between /* ... */.

DIGIT	:	[0-9]
INT	:	DIGIT+
LETTER	:	[a-z | A-Z]
ID	:	LETTER (LETTER | DIGIT)*
SPECIAL-SYMBOL	:	**,** | **.** | **..** | **:** | MATH-OP | BOOLEAN-OP | IMPLIES | BRACKETS
MATH-OP	:	+ | − | * | / | % | ^ | :=
BOOLEAN-OP	:	= | > | >= | < | <= | & | |
IMPLIES	:	− >
BRACKETS	:	**{** | **}** | (|) | [|]
KEYWORDS	:	**system** | **end** | **component** | **motif** | **type** | **initial** | **from** | **to** | **on** | **port** | **int** | **boolean** | **motif** | **for** | **each** | **where** | RECONFIG-ACTION | LITERALS | OTHER-OP
RECONFIG-ACTION	:	**create** | **join** | **leave** | **delete**
LITERALS	:	**true** | **false** | **null**
OTHER-OP	:	**not** | **max** | **min** | **avg** | **random** | **dist** | **sin** | **angle** | **cos** | **indexOf** | **size**
WS	:	[\t \r \n]+
COMMENTS	:	// .*? (\r | \n)
COMMENT	:	/* .*? */

Table 5.1: Lexical structure of the Dr-BIP language

5.2.2 Grammar Highlights

This section presents the grammar that specifies how a Dr-BIP system is defined by a system designer. More precisely, a grammar is a set of production rules that defines how lexems introduced earlier can be arranged to create a Dr-BIP system description. It specified how lexems can be grouped to describe a component type, motif type, interaction rule, and reconfiguration rule etc. Each production rule is of the form S $\rightarrow$ w, where S is a nonterminal symbol and w is a string of terminals and/or nonterminals or epsilon. A terminal symbol typically refers to a lexeme and a nonterminal symbol refers to a production rule.

Table 5.2 highlights the prominent grammar rules. To simplify grammar rules we utilize regular expression symbols *, +, and ? to categories the appearance count of a nonterminal symbol. For example, the rule S $\rightarrow$ w* implies that w can occur zero or more times and is a short hand for the rules S $\rightarrow$ w and w $\rightarrow \beta$w | ϵ where β is a terminal symbol. Moreover, the rule S $\rightarrow$ w+ implies that w can occur one or more times and is a short hand for the rules S $\rightarrow$ w and w $\rightarrow \beta$w | β. Finally, the rule S $\rightarrow$ w? implies that w can occur zero or one times and is a short hand for the rules S $\rightarrow$ w and w $\rightarrow \beta \mid \epsilon$. We elaborate next only on the relevant grammar rules that define a Dr-BIP system, component and motif types, interaction rules, and reconfiguration rules.

Dr-BIP System. A Dr-BIP system description is composed of a nonempty list of component type definitions, followed by a nonempty list of motif type definitions. furthermore, it is optionally followed by a list of inter motif reconfiguration rules. Each system has a name defined by its ID. The corresponding grammar rule describing the syntax of a system is labeled by *system*, and is defined in Table 5.2. Therefore, system description has the form:

```
system ID
  <component_type_list>
  <motif_type_list>
  [<inter_reconfiguration_rule_list>]
end
```

where the notation [..] denotes an optional construct.

Component Types. A component type defines a nonempty list of ports and its behavior. Each component type will have a name specified with its ID. Moreover, each component type optionally maintains a list of data. The corresponding grammar rule, labeled by *componentType*, is defined in Table 5.2. Therefore, a component type definition has the form:

system	→	**system** *ID componentType+ motifType+ interReconfigurationRule** **end**
componentType	→	**component type** *ID data* port+ behavior* **end**
behavior	→	initTransition transition*
data	→	*type ID* (**,** *ID*)*
type	→	(**int** | **boolean**)
port	→	**port** *ID* (**,** ID)*
initTransition	→	**initial to** *ID guardActionPair*
guardActionPair	→	*guard* → *action*
guard	→	[expr]
action	→	**{** expr (COMMA expr)* **}** | **{ }**
transition	→	**on** *ID* **from** *ID* **to** *ID guardActionPair*
motifType	→	**motif type** *ID componentInstance+ rules** **end**
componentInstance	→	**component** *ID ID* [expr]?
rules	→	*interactionRule* | *reconfigurationRule*
interactionRule	→	*nested-iterators*? **interaction** id^+ **where** *portAssign guardActionPair*
nestedIterators	→	*iterator+*
iterator	→	*forEachIterator* | *forIterator*
forEachIterator	→	**for each** *ID ID* **:** *expr*?
forIterator	→	**for** ID **in** **{***expr* **..** *expr* **}** **:** expr? *guardActionPair*
reconfigurationRule	→	*nestedIterators guardActionPair*
expr	→	*expr-literal*
	|	*expr-variable*
	|	*expr-unary*
	|	*expr-binary*
...		

Table 5.2: Highlight of Dr-BIP grammar rules

```
component type ID
  [<data_list>]
  <port_list>
  <behaviour>
end
```

A component type may define a list of data which holds its attributes. Each data has a type and a name defined by its ID. A data type can either be an integer (**int**), or a **boolean** (boolean). The corresponding grammar rule labeled by *data*, in Table 5.2, defines a set of data having the same type. Therefore, A component

data definition has the form:

```
<type> ID [, ID ...]
```

A component type also defines its ports which are dedicated to interact with other components. Each port is defined by its ID. The grammar rule labeled by *port* defines. The port names for a component type use the form:

```
port ID [,ID ...]
```

A component type defines its behavior in a form of a transition system. A component type will have multiple states, one of which is the initial state determined by its initial transition. Transitions are defined from one state to the other. Each transition has a *guard* expression which determines when the corresponding transition is enabled. In addition, each state transition rule defines a set of *action* expressions which are executed after a successful execution of the state transition rule. The corresponding grammar rules labeled by *initTransition* and *transition* define respectively describe the syntax of an initial transition and a transition which together dictate the behavior of a component type. The grammar rules are defined in table 5.2. Therefore, a component behavior definition has the following form, where each ID is an identifier representing a state name.

```
initial to ID [<expr>] -> {<expr> [, <expr> ...]}
on ID from ID to ID [<expr>] -> {<exp> [, <expr> ...]}
on ID from ID to ID [<expr>] -> {<exp> [, <expr> ...]}
...
on ID from ID to ID [<expr>] -> {<exp> [, <expr> ...]}
```

Motif Types. The current prototype sets restrictions on maps and deployments. Maps are restricted to simple graphs e.g., chain, cyclic, star. To achieve this, the current implementation utilizes array lists to aid the representation of a map. Component instances can be deployed on an array list using indexes. This represents the deployment. Various indexing patterns can be used to simulate different simple map like structures such as a ring, line, square etc.

A motif type is defined by a nonempty list of component instances and an optional list of rules. A rule can either be an interaction or a reconfiguration rule. Moreover, each motif type has a name defined by its ID. The corresponding grammar rule defining the syntax of a motif type is labeled by *motifType* and is defined in Table 5.2. Therefore a motif type definition has the form:

```
motif type ID
  <component_instance_list>
```

```
  [<rules_list>]
end
```

A component *instance* must refer to a component type which dictates its behavior, data and ports. A component instance is therefore defined by two ID's the first ID refers to the name of its component type and the second refers to the name of the component instance itself. A component instance can either be a typed array list of component instances, where all the elements in the array are component instances of the same type or a singleton component instance. In the case of a typed array, an expression determines the size of the array. A component instance array list definition has the form:

```
component ID ID[<expr>]
```

Where as a single component instance definition has the form:

```
component ID ID
```

Interaction Rules. An interaction rule dictates possible interactions between two or more components. It utilizes iterators to enable the definition of multiple interactions with a single rule. The grammar rule corresponding to the definition of an iterator is labeled by *iterator* in Table 5.2. The interaction rule first defines an optional list of iterators, followed by the set of involved ports of each participating component followed by a guard expression that must also evaluate to true for an interaction to be enabled. Hence an interaction is enabled if the ports of its involved components are enabled and if the guard expression evaluates to true. Finally, an interaction rule defines an optional list of action expressions. These action expression must be executed upon the completion of an interaction rule. The grammar rule corresponding that defines an interaction rule is labaled by *interactionRUle* in Table 5.2. An interaction rule definition has the form:

```
[<iterator_list>]
interaction ID .. ID
where ID := ID.<portID> [, ID := ID.<portID> ...]
[<expr>] -> {[ <expr> ...]}
```

Reconfiguration rules. Similar to interaction rules, reconfiguration rules define an optional list of iterators that facilitate the definition of multiple reconfigurations within a single rule. The list of iterators is followed by a guard expression and an optional list of action expressions. Action expressions in a reconfiguration

rule differ than those in an interaction rule in that they allow the creation/deletion/migration of components/motif. The corresponding grammar rule defining a reconfiguration is labeled by *reconfigurationRule* in Table 5.2. A reconfiguration rule defintion has the form:

```
[<iterator_list>]
[<expr>] -> {[ <expr> ...]}
```

5.2.3 An Example Using The Concrete Syntax

In this section, we present an example of a system description/code that conforms to the concrete syntax. The code presented below reflects the source file presented in Figure 5.1. Dr-BIP code is organized into systems. Each *system* must be contained in a single source file. Multiple systems may not be defined in the same file. The example below demonstrates the code for the self-configuring ring system example presented in section 4.1.

In summary, a self-configuring ring system has one component type *Element* and a *Ring* motif type. A ring motif type is initialized with three element. Two elements compose the ring structure, and the third element is a dummy element that is there to get the token passing started. The motif type has one interaction rule for token passing and two reconfiguration rules to respectively add and remove an element from the ring. Finally, the system has one inter-motif reconfiguration rule that merges two rings together.

```
1  system selfConfiguringRing
2    component type Element
3      boolean idle
4      port inPort, outPort
5
6      initial to q0 [true] -> {idle := true}
7      on inPort from q0 to q1 [true] -> {idle := false}
8      on outPort from q1 to q0 [true] -> {idle := true}
9    end
10
11   motif type Ring
12     component Element elements[2]
13     component Element dummy[1]
14
15     // interaction rule get ring started
16     interaction inn1 inn2
17     where inn1 := dummy.inPort, inn2 := elements[0].inPort
```

```
18      [true]->{}
19
20      //interaction rule for token passing
21      for i in {0 .. elements.size-1}:
22      for j in {0 .. elements.size-1}:j = mod(i+1,elements.size)
23      interaction out inn
24      where out := elements[i].outPort, inn := elements[j].inPort
25      [true] -> { }
26
27      // delete an element in ring
28      for i in {0 .. elements.size-1}:
29      [(elements.size > 2) & (elements[i].idle = true)] ->
30      {delete(elements[i])}
31
32      // create a new element in Ring
33      for i in {0 .. 0}:
34      [true] -> {create(Element, temp), join(temp, elements)}
35
36    end
37    //inter motif reconfiguration merge rings
38    for each Ring r1:
39    for each Ring r2: not(r1 = r2) &
40              ((r1.elements.size  + r2.elements.size) < 10)
41    [not(r1.elements in r2.elements)] ->
42              { join(r1.elements,r2.elements),
43              leave(r1.elements, r1.elements),
44              delete(r1)}
45  end
```

5.3 Parser

The previous sections presented the first phase of the prototype implementation. They introduced the concrete syntax, and an example of a system description conforming to it. This section presents the second phase of the prototype implementation, the parsing phase. In a nut shell, the parser takes as input a system description/code that conforms to the concrete syntax and produces an instance of a Dr-BIP model.

To build the parser we used Antlr 4 [87], a powerful language recognition framework. The parsing is completed with three steps. First, the input stream is

scanned one character at a time and characters ar grouped into lexemes that are defined by the lexical structure in section 5.2.1. Next, the stream of lexemes is scanned to build an abstract syntax tree (AST) based on the grammar defined in 5.2.2. Figure 5.2 illustrates a partial view of an abstract syntax tree that results from parsing the example presented in section 5.2.3. Its worth noting that in order to generate a parse tree, direct and indirect left recursions found in grammar rules have been eliminated. This is because left recursion can lead to infinite parsing specifically in a top-down parser. Finally, the AST tree is traversed to create a model instance that corresponds to the input code.

A model instance is a JAVA object of type system that has as attributes a list of component and motif types and a bunch of motif and component instances etc. Traversing the AST typically calls for the creation of an object instance at each node, where the object type corresponds to the node. For example when the node in the AST refers to an expression an Expression object is instantiated. Therefore, all the framework concepts have been encapsulated into classes in JAVA enabling the creation of a model instance.

To simplify testing we introduce input parameters that specify how many motif instances are there initially of each motif type. Parameters allow the creation of multiple model instances of the same system but with varying initial configurations. For example, parameters can be used to generate a self-configuring ring model with 10, or 50 rings. Therefore, the parser takes as input both a system description file and parameters in order to generate a model instance.

5.4 Interpreter

The operational semantics introduced in chapter 3 describe the behavior rather than the structure of a system model. They provide a meaning to the modeling language concepts and thus, supports the interpretation of models expressed in the language. In other words, the interpretation gives the model a meaning by mapping its language concepts to the semantic domain. Consequently, the semantics are implicitly encoded in the implementation of an interpreter. The interpreter corresponds to the last phase of the prototype implementation.The interpreter is developed in JAVA. In a nut shell, it allows the computation of enabled interactions and reconfigurations in a model instance, and their execution according to predefined policies (interactive, or random).

Initially, the interpreter requests the status of the model instance. The model instance status is composed of the status of all its motif instances and the status of their constituting component instances. Once the interpreter has a current view of the system, it computes the enabled interactions and reconfigurations by evaluating the corresponding rules in accordance with the semantics defined earlier.

Next, the interpreter nondeterministically picks a step i.e. either a reconfiguration or an interaction to be executed. The model instance is updated in accordance with the chosen step. For example, if the chosen step is an interaction, the involved motif and participating components are informed and the interaction is executed. The execution of an interaction implies the execution of the corresponding actions followed by a state change for participating components determined by their behavior. If the chosen step is a reconfiguration, then the involved motifs are notified and reconfiguration associated actions ar executed.

In summary, given the current system view the interpreter evaluates the interaction and reconfiguration rules at every computational step and decides on the allowed steps. By default the interpreter will choose randomly a step to execute and will execute 100 such steps, unless specified otherwise by input parameters. We discuss parameters in details next.

5.4.1 Parameters

To facilitate and increase the flexibility during testing we introduce parameters that alter the default execution of the interpreter. These parameters are bound at runtime and can be defined using command options discussed in the following section. There are three main parameters. The first one alters the number of runs executed. This is practically useful for interpreting multiple instance models of the same system. The second parameter alters the number of steps executed in each run. The default value for this parameter is set to 100. The third parameter alters the selection choice for the upcoming step. By default, the interpreter nondeterministically choses the next step, however this could be changed so that the next step is chosen interactively by the user.

5.5 Execution

The parser and interpret are grouped into a single jar file. Running the jar file will therefore require a JAVA run time environment to be installed on the system. The latest Java Runtime can be found on www.oracle.com. The following command is the most general one that will execute with predefined default values.

```
java -jar drbip.jar -i <file path>  -m <motif type name>
  <instance count>
```

. The command will direct parser to create a model instance from the the input file specified. The model instance will have a number of motif instances as specified by the command. Next the interpreter will perform a single run over the model

instance and it nondeterministically choses the next 100 steps. It is possible to cater the command with the options presented in the table below.

Options	Description
-i, - -input <file path>	define the input file path
-m, - -motif <motif name> <instance count>	sets the number of instances to instantiate for a motif type
-r, - -runs <run count>	specify the number of runs the system makes, the default value is set to 1
-s, - -steps <step count>	specify the count of steps in each run, the default value is set to 100
-n, - -nondeterministic <boolean value>	system picks random interaction if true, otherwise user picks the next interaction to be executed
-d, - -debug	prints debugging logs
-h, - -help	prints help message

Table 5.3: Description of options available to cater the execution of the prototype

For example, to make 1000 steps while interactively picking the next step, instead of having the next step nondeterministically chosen use the command below.

```
java -jar drbip.jar -i <file path> -m <motif type name>
  <instance count> -s 1000 -n false
```

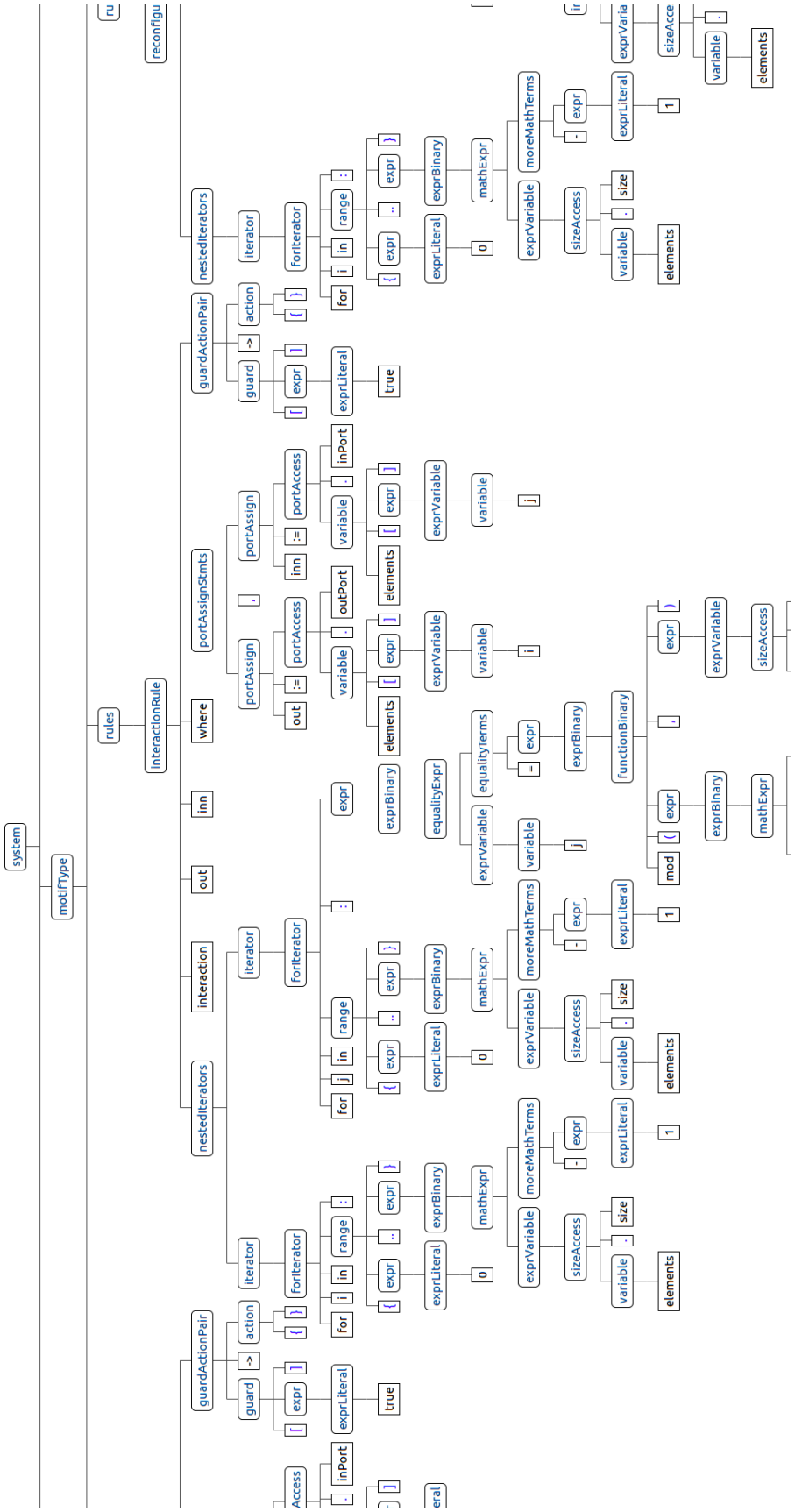


Figure 5.2: Partial view of an abstract syntax tree

Part III

Conclusions and Perspectives

Chapter

6

Conclusion and Perspective

This chapter concludes the thesis with a summary for each part, and a peak into the future identifying plausible potential directions.

Conclusion

Part 1. Modern systems are pressured to adapt in response to their constantly changing environment to remain useful. Traditionally, this adaptation has been handled manually and at down times of the system. The cost of such an adaptation has been steadily increasing and is estimated to be more than 90% of the total cost of the entire system's life cycle [3]. There is an increased demand to automate this process and to achieve it whilst the system is running. *Self-adaptive systems* were introduced as a realization of continuously adapting.

We surveyed existing interpretations for the term self-adaptive systems in the literature, we highlighted discrepancy and similarities in the definitions. We concluded with two major school of thoughts. In the first, adaptation is triggered in response to change, while in the second adaptation is triggered when the system is not achieving its goals and requirements. This disagreement in the literature on a single definition results in contrasting system design. In the first school of thought, systems will monitor things that change which typically include the environment, system, or requirement. In the second school of thought, the system will

most likely monitor a quantification measure of how well the system achieves its requirements. We ally our proposal with the first school of thought where systems adapt at runtime to changes in themselves and their environment.

Next, We presented the requirements of a self-adaptive system in the form of a conceptual model. A self-adaptive system is typically composed of three elements an adaptation engine, managed system and environment. The adaptation engine supervises and administrates the managed system. The adaptation engine is responsible for monitoring and affecting adaptation in the managed system and the environment. It does so by relying on a closed MAPE-K loop, where it *m*onitors and collects data from both the managed system and the environment, after which it *a*nalysis the collected data. Next, it decides on an adaptation *p*lan, which it *e*xecutes later on. The managed system comprises the system's main functionality and it accommodates actuators which enable its adaptation by the adaptation engine. Together these three elements orchestrate self-adaptation in systems.

Self-adaptive systems are known to have four main properties, which are self-configuration, self-optimization, self-healing, and self-protection. Designing and modeling self-adaptive systems that realize all these properties remains an interesting challenge to be explored in future research. Self-configuration was the main property of interest in this work as it is an essential property. Self-configuration is the capability of reconfiguring automatically and dynamically in response to changes. This may include installing, integrating, removing and composing/decomposing system elements. The importance of self-configuration is brought to light by understanding that it is essential to attain the remaining properties. For example, a system could isolate an infected element in order to self-protect itself. Similarly, a system may require the installation of new elements to handle high system load i.e. self-optimize.

The process of realizing self-adaptive system poses various challenges that we introduce in relation to the conceptual model. Some of these challenges include developing a framework that is generic enough to tackle problems in various domain, yet expressive enough to model complex problems. Another challenge is picking the right level of abstraction when modeling the managed system. The preliminary question to answer is how much to abstract away from the running system without loosing the system's integrity? Moreover, It is not only important model dynamically adapting systems but also to have the ability to assure that their new and transient behavior conforms to the system's safety property. In part 2, we show how we addressed some of these challenges.

Consequently, we presented a road map of various approaches used in the literature to realize self adaptive system. In addition, We highlighted the advantages and disadvantages of each. For instance, developing a self-adaptive system from a control engineering perspective implies designing a control-based self-adaptive

system whose behavior can change according to a set of well-formed mathematical models that can be formally analyzed. Most of the mentioned approaches are used in combination with others to achieve self-adaptation. For instance, the interplay of the model-based and component & connector approach sets the backbone of our framework. On the one hand, a model-based approach to self-adaptation employs runtime models as a representation of the running system. Such an approach endorses, automatic reasoning, and system monitoring. On the other hand, a component & connector based approach focuses on the architecture of the underlying system and hence encapsulation, abstraction, reusability, and scalability are primary advantages of such an approach. In a component & connector approach, the system description is composed of components and connectors. Components encapsulate functionality of the systems and connectors dictates the inter- action between components.

Part 2. We proposed the DR-BIP framework as a solution for modeling self-configuring adaptive systems. Conceptually, the framework is composed of a centralized engine and a runtime system model. The engine administrates the system model. It determines the need for a reconfiguration/interaction and reflects the chosen adaptation in the system model. The system model is a representation of the running system at three different levels of abstraction, namely behavior, configuration and configuration variant. We introduced architectural motifs to capture the three levels of abstractions. A motif encapsulates (i) behavior, as a set of components, (ii) interaction rules and (iii) reconfiguration rules. A system model is a superposition of motifs. Describing systems as superposition of motifs endorses enhanced flexibility and abstraction where, each motif alone is a self-configuring architecture with its own coordination rules i.e interaction and reconfiguration rules. Hence the membership of a component in a motif determines the way it interacts with other components and the reconfiguration rules it is subject to.

We introduced behavioral types (component types) to facilitate the description of complex and large systems with multiple elements exposing the same behavior. Behavioral types allow the creation of several elements of a certain behavior (i.e. type). For instance, introducing new servers to address high load. More importantly, behavioral types allow the definition of parametric interactions, where a single interaction rule can define multiple interactions based on behavioral types for a given motif.

A component in a motif must therefore refer to a component type which defines its behavior. To structure components in motifs, we proposed maps and deployment. Maps are graph-like structure consisting of interconnected positions. Deployments relate components to positions on the map. Maps are used to naturally express mobility and dynamically changing environments. They prove to be very

useful for both the parametrization of interactions and the mobility of components. It is important to note that a map can have either a purely logical interpretation, or a geographical one or a combination of both. For instance, a purely logical map is needed to describe the functional organization of the coordination in a ring or a pipeline. A map with geographical interpretation is needed to describe mobility rules of cars on a highway. Such a map will represent the structure of highway with fixed and mobile obstacles. Finally, a map with both logical and geographic connectivity relations is needed to express coordination rules for cars on a highway, where coordination rules depend not only on the physical environment but also on the logical communication features available.

Dr-BIP supports self-configuration at three different levels of granularity, namely components, connectors and subsystems. It supports the explicit addition and removal of components and subsystems (i.e. motifs). However the addition and removal of connectors is implicit. The main advantage of relying on an implicit addition and removal of connectors is the ability to guarantee by construction specific configuration topologies. When a new component is added to the system, its interaction is implicitly defined depending on its type and predefined coordination rules of the motif it belongs to. Therefor, there is no need to explicitly specify interactions/connectors for new components. Moreover, the interaction of existing components in the system can implicitly change in two ways. First, through the migration of components to other motifs that define different interaction rules. Second, through the dynamic change of the deployment of a component. The second way is possible only if interaction rules are dependent on connectivity rules in the map.

The conceptual model of the Dr-BIP framework is similar to the generic conceptual model proposed in part 1 for self-adaptive systems. The main difference lies in modeling the system's environment. The system model implicitly captures the environment with the aid of reconfiguration rules. For example, when a server is experiencing a high load as a result of many user requests (environment) it is expected to adapt by delegating tasks to another server. To handle this in Dr-BIP a reconfiguration rule is defined that migrates tasks to other servers when a server's load reaches a certain value. In conclusion, the system's environment must be studied thoroughly in advance, and corresponding reconfiguration rules must be defined to account for and respond to any environmental change.

To summarize the Dr-BIP framework is designed to address some of the challenges mentioned in part 1 by ensuring that the framework is built with specific design pillars in mind. First and foremost, Dr-BIP endorses generality by relying on common, but effective, architecture abstractions such as component and connectors to model systems. Such abstractions are generic and consequently they are applicable to a wide range of domains. Moreover, Dr-BIP relies on a

well-defined operational semantics. Its semantics leverage on existing models for rigorous component-based design (from its predecessor BIP). In addition, Dr-BIP can guarantee by construction specific properties through the definition of configurations as parametric operators on components. Finally, One of the main assets of the framework is separation of concerns. Dr-BIP relies on exogenous global coordination rules which allows to reason separately about the system function and its adaptive behavior. To the best of our knowledge, there is no exogenous coordination language such as an ADL addressing all these modeling issues in such a methodologically rigorous manner.

We demonstrated that the proposed framework is minimal and expressive allowing concise modeling with the aid of four example. Each example was presented by first defining the intended target system behavior followed by its corresponding modeling in Dr-BIP . We showed that Dr-BIP is designed with autonomy in mind with the examples on autonomous highway traffic system and Self-organizing robot colonies that demonstrate the power of its structuring concepts (motifs). Designing systems as a superposition of motifs with their own coordination rules tremendously simplifies the description of autonomous behavior.

Last but not least, as a proof of concept we developed a prototype implementation with restrictions on maps and deployments. Maps are restricted to simple graphs e.g., chain, cyclic, star. We first defined the concrete syntax of the modeling language with the aid of lexical structure and grammar rules, after which we built a top-down parser using both Antlr and JAVA. The parser takes as input a system description code, conforming to the concrete syntax, and outputs an instance of the system model. Technically, the model instance is a JAVA Object representing a Dr-BIP system. Finally we built an interpreter, also in JAVA, which takes as input a model instance and outputs a trace summarizing the executed steps. In a nut shell, the interpreter will compute the enabled interactions and reconfigurations for a given model instance, decide on an enabled step (either an interaction or reconfiguration), and assure the execution of the chosen step by the model instance.

Perspective

We categorize future work by elements of the framework's conceptual model proposed in section 3.1.2, namely the engine and system model. We propose potential extension of the engine and system model, after which we suggest the addition of a tool-set to accompany the framework.

Engine

Decentralization. Decentralization of the engine is an essential expansion that is specifically useful for modeling self-configuring complex systems whose main requirement is scalability. Decentralization is mainly concerned with how adaptation decisions in a self-adaptive system are coordinated regardless of the distribution the system. Typically, if the system is deployed on a single location, then the adaptation engine is most likely to be centralized. In a centralized approach, the engine maintains a global view of the system and has full control over it, which simplifies the adaptation decisions making. However this is not suitable for large systems because of the size and real time constraints. Similarly, when the system is distributed, it is most likely that the adaptation engine is decentralized.

IOT systems are an example of self-configuring systems that intrinsically require a decentralized engine, as their system elements are most likely to be distributed and deployed at various locations. It is therefore important to address this issue to facilitate the adoption of Dr-BIP by industry. Most of existing work in the literature focuses on a centralized self-configuring engine, that is partly because decentralization poses interesting new challenges. For instance, the need arises for effective coordination and communication between engines. Moreover, dependencies among system elements poses another dimension of complexity. These issues are more prominent in endogenous coordination models where the adaptive behavior and non-adaptive are mixed together. For example, reconfiguration description can be specified within the component behavior description. In an endogenous coordinated system global coordination mechanisms do not have to be defined, but this introduces dependencies between components.

The Dr-BIP framework with its exogenous coordination and its structuring constructs (motifs) mitigates away from the complexity of decentralization. Each motif maintains its coordination rules, and as such each motif can evolve alone according to its rules. Therefore, decentralization naturally manifests itself in a motif-based system. However there are still important issues to address that arise as a result of shared components even in the simplest form of decentralization, namely the *fully decentralized* approach. In a fully decentralized approach each subsystem (i.e. a motif) has its own engine. Such decentralized coordination is organized as follows: each motif's engine collects the status of its motif's constituent components, communicates with other motifs to collect up to date knowledge about shared components, computes the enabled interaction and reconfiguration according to its rules, coordinates with other engines when needed to synchronize their actions.

Such a decentralization requires effective communication across engines to share knowledge, which may introduce latency. Latency can lead to inconsistent views of the system. To combat this and reduce latency we consider the situations

that will inevitably require a communication across motifs which are two. The first scenario involves a regional interaction i.e an interaction which involves some shared component. The second scenario is a inter-motif reconfiguration involving two motifs. Therefore, local interaction (involving only exclusive components in a motif) or reconfiguration may be executed without requiring communication with other motif engines. It might also be appropriate to consider a hybrid approach, which combines both centralized and decentralized to combat latency. The authors in [17] investigate the different patterns in decentralizing the adaptation engine. This work could be a kick starting point.

System Model

History variables. One of the main benefits of the framework is separation of concerns which keeps separate the component behavior from its coordination. Moreover it keeps its interaction coordination separate from its reconfiguration coordination. The dichotomy between interaction and reconfiguration steps ensures separation of concerns for execution within a motif as previously discussed in section 3.2.1 and illustrated in Figure 3.7. However this separation of concerns comes with some consequences. Consider a reconfiguration action that requires some form of multi party interaction among components. For instance, in the platoon example in section 4.3, the split action is defined by the target system as: a request from an arbitrary car to leave the platoon and when the split is performed the leading platoon will increase its speed by 2% whereas the tail platoon will reduce its speed by 2%. Due to the separation of concerns, this is modeled in Dr-BIP with two separate rules. The first rule is an interaction rule labeled by *sync-platoon-split* which sets the new speed for both platoons. The second rule is a reconfiguration rule labeled by *do-platoon-split* which reconfigures the motif and splits the platoon.

Logically, the split reconfiguration step must be enabled only if the split interaction has been executed. This calls for some form of relation or dependency between interaction and reconfiguration that is not captured by the current framework. In fact, the split reconfiguration is always enabled in the current framework. To solve this issue we propose the use of history variables that keep track of the executed interactions. In addition, we can redefine reconfiguration rule constraints to include history variable constraints that comply a specific interaction to be in the history variable for a reconfiguration to be enabled.

Priorities. In the previous section we proposed the use of history variables to define dependability between reconfigurations and interactions. History variables have solved the issue of enabling the split reconfiguration only when needed,

however there is still another issue to consider here. Assume that a platoon split interaction has been executed, then (with the aid of history variables) this will enable the corresponding split reconfiguration. However, keep in mind that other steps might be enabled too and since the engine picks the next step nondeterministically, if the engine executes the *sync-platoon-split* interaction, there is no guarantee that the corresponding split configuration will be executed immediately after.

Therefore it is essential to have some form of priority policy that will filter out other enabled steps in such a situation. A priority policy defined as a partial order between the steps, will put some enabled steps in favor over other enabled steps. Therefore, at any given state, the engine executes only the steps with maximal priority amongst those currently enabled according to the interaction and reconfiguration rules. For instance, we can define the priority policy for the platoon example in such a way that split reconfiguration is given higher priority over all other steps (interaction and reconfigurations). Hence, given multiple platoons that performed a split interaction, the split reconfiguration step for all platoons will be of highest priority, and the engine will pick nondeterministically between them.

Framework

Toolset Extension. We have shown the validity of the framework with four examples, however this is not sufficient. Verification is a primary requirement of self-configuring systems as it is important to verify that the execution of reconfiguration satisfies system's nonfunctional properties such as liveness, deadlock freedom, and safety. For instance, it is essential to verify that the integration of new components does not introduce deadlock. Therefore future work aims at extending the framework with a verification toolset that will unlock the potential of Dr-BIP beyond the academic setting.

In [88], the authors survey 75 studies for the use of formal methods in self-adaptive systems. They inspect the various modeling languages used for formalization including transition system, markov models, petri nets, and process algebra etc. Moreover, they survey the specification languages used by these studies to formally specify system properties including graph grammar, logic (first order logic and linear temporal logic etc.), and others. Finally they inquiry the different system properties that are verified. They conclude with the increase attention of formal methods in the domain of self-adaptive system, however despite this increase the absolute number of studies employing formal verification remains low. That is partly because of the challenges that arise as a result of extending traditional verification techniques such as model checking to self-adaptive systems.

Model checking is one of the prominent strategies that automatically checks whether a given property is met by a system model through exploring its state

space. Self-adaptive system's state space can grow quit large in size due to the inherent nature of these systems i.e. having several possible alternatives. Therefore, Model checking can be infeasible for self-adaptive systems due to the state-space explosion. This is why [88] proposes a way to dissect the state space of self-adaptive systems through the introduction of *zones*. The zone based approach presents an interesting starting point to better understand modeling checking of self-adaptive systems. They propose four zones normal behavior, adaptation behavior, undesired behavior, and invalid behavior. The normal behavior zone includes the set of states where the system is performing its domain functionality. The adaptive behavior zone includes the set of states where the system is recovering from an undesired behavior (state). The undesired behavior zone is the set of states where the system requires adaptation due to some concern such as failure etc. The invalid behavior zone corresponds to states that the system must not reach such as a deadlock state. In other words it represents the behavior that the system should not exhibit. They claim that various system properties such as liveness and deadlock map to transitions between these four zones. The aforementioned work and [89] are good starting point for a holistic approach for verification of Dr-BIP .

For the time being, it is possible reason about verification of a Dr-BIP system in a compositional manner. Naturally, the non-configuring parts of the system (i.e. motif-based system without reconfiguration rules) can be verified using traditional verification techniques levering from static BIP. To verify correctness of the parametric interacting system with the assumption that dynamic connectors correctly enforce the sought coordination, it remains to show that restricting the behavior of deadlock-free components does not introduce deadlocks. To achieve this, the DFinder [90] can be extended for parametric systems which requires the solution of parametric Boolean equations. D-Finder uses compositional approach to verify component-based systems described in static BIP. DFinder implements deadlock detection by computing incrementally stronger invariants and applying proof strategies to eliminate potential deadlocks.

Given that we have proven the correctness of the parametric interacting motif-based system, verifying the correctness of reconfiguration operations remains a process of proving that the motif's architecture style is preserved by reconfiguration actions which modify the number of components, their connectivity, their deployment, and maps. Therefore, the architecture style can be seen as in invariant of the coordination structure. This can be proven by structural induction, where the architecture style of a motif can be characterized by a configuration logic formula ϕ as in [91]. Therefore, we have to prove that if a model m of the system satisfies ϕ, then after the application of a reconfiguration operation the resulting model m^1 satisfies ϕ.

Appendix

A

Appendices

A.1 Self-adaptive System Definitions

Self-adaptive systems have been defined differently in the literature. This list highlights the similarities and differences in defintions. A compact comparison is presented in section 1.2.1. Self-adaptive systems are:

> “ *systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their requirement* ”
>
> **De Lemos et al. [5]**

> “ *[systems that are able to] adjust various artifacts or attributes in response to changes in the self and in the context of a software system. By self, we mean the whole body of the software, mostly implemented in several layers, while the context encompasses everything in the operating environment that affects the system's properties and its behavior* ”
>
> **Salehie and Tahvildari [6]**

" *systems that are able to adjust their behaviour in response to their perception of the environment and the system itself* "

Cheng et al. [8]

" *[systems that] modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation* "

Oreizy et al. [9]

" *systems [..] which can modify their behavior at run-time due to changes in the system, its requirements, or the environment in which it is deployed* "

Andersson et al. [10]

" *[systems] which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system* "

Kramer and Magee [11]

" *systems endowed with the ability to respond to a variety of changes that may occur in their environment, goals, or the system itself by adapting their structure and behaviour at run-time in an autonomous way* "

Cámara et al. [12]

" *systems that respond to change by evolving in a self-managed manner while running and providing service* "

Calinescu et al. [13]

" *[systems that] dynamically adapt its behavior at run-time in response to changing conditions in the supporting computing, communication infrastructure, and in the surrounding physical environment* "

Zhang and Cheng [14]

" *[systems that] can identify, decide and perform required activities appropriately in situation for software context responding to sophisticated hardware and difficult prediction for various change properties* "

Cha, Kim, and Jeong [15]

Bibliography

[1] Rob Kitchin and Martin Dodge. *Code/space: Software and everyday life*. Mit Press, 2011.

[2] Autonomic Computing et al. "An architectural blueprint for autonomic computing". In: *IBM White Paper* 31 (2006), pp. 1–6.

[3] Len Erlikh. "Leveraging legacy system dollars for e-business". In: *IT professional* 2.3 (2000), pp. 17–23.

[4] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50.

[5] Rogério De Lemos et al. "Software engineering for self-adaptive systems: A second research roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[6] Mazeiar Salehie and Ladan Tahvildari. "Self-adaptive software: Landscape and research challenges". In: *ACM transactions on autonomous and adaptive systems (TAAS)* 4.2 (2009), p. 14.

[7] Javier Cámara et al. "Adaptation impact and environment models for architecture-based self-adaptive systems". In: *Science of Computer Programming* 127 (2016), pp. 50–75.

[8] Betty H. C. Cheng et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–26. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_1. URL: https://doi.org/10.1007/978-3-642-02161-9_1.

[9] Peyman Oreizy et al. "An architecture-based approach to self-adaptive software". In: *IEEE Intelligent Systems and Their Applications* 14.3 (1999), pp. 54–62.

[10] Jesper Andersson et al. "Modeling dimensions of self-adaptive software systems". In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 27–47.

[11] Jeff Kramer and Jeff Magee. "Self-managed systems: an architectural challenge". In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, pp. 259–268.

[12] Javier Cámara et al. "Testing the robustness of controllers for self-adaptive systems". In: *Journal of the Brazilian Computer Society* 20.1 (2014), p. 1.

[13] Radu Calinescu et al. "Self-adaptive software needs quantitative verification at runtime". In: *Communications of the ACM* 55.9 (2012), pp. 69–77.

[14] Ji Zhang and Betty HC Cheng. "Model-based development of dynamically adaptive software". In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 371–380.

[15] Jung-Eun Cha, Jeong-Si Kim, and Young-Joon Jeong. "Architecture Based Approaches Supporting Flexible Design of Self-Adaptive Software". In: *Computational Science and Computational Intelligence (CSCI), 2016 International Conference on*. IEEE. 2016, pp. 1424–1425.

[16] Danny Weyns. "Software engineering of self-adaptive systems: an organised tour and future challenges". In: *Chapter in Handbook of Software Engineering* (2017).

[17] Danny Weyns et al. "On patterns for decentralized control in self-adaptive systems". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.

[18] Mazeiar Salehie and Ladan Tahvildari. "Towards a goal-driven approach to action selection in self-adaptive software". In: *Software: Practice and Experience* 42.2 (2012), pp. 211–233.

[19] David Garlan et al. "Rainbow: Architecture-based self-adaptation with reusable infrastructure". In: *Computer* 37.10 (2004), pp. 46–54.

[20] Christian Krupitzer et al. "A survey on engineering approaches for self-adaptive systems". In: *Pervasive and Mobile Computing* 17 (2015), pp. 184–206.

[21] Danny Weyns, Sam Malek, and Jesper Andersson. "FORMS: Unifying reference model for formal specification of distributed self-adaptive systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7.1 (2012), p. 8.

[22] Jacqueline Floch et al. "Using architecture models for runtime adaptability". In: *IEEE software* 23.2 (2006), pp. 62–70.

[23] Danny Weyns, M Usman Iftikhar, and Joakim Söderlund. "Do external feedback loops improve the design of self-adaptive systems?: a controlled experiment". In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press. 2013, pp. 3–12.

[24] Shang-Wen Cheng and David Garlan. "Stitch: A language for architecture-based self-adaptation". In: *Journal of Systems and Software* 85.12 (2012), pp. 2860–2875.

[25] Carlos Cetina et al. "Autonomic computing through reuse of variability models at runtime: The case of smart homes". In: *Computer* 42.10 (2009).

[26] Mathieu Acher et al. "Modeling context and dynamic adaptations with feature models". In: *4th International Workshop Models@ run. time at Models 2009 (MRT'09)*. 2009, p. 10.

[27] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. "FUSION: a framework for engineering self-tuning self-adaptive software systems". In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM. 2010, pp. 7–16.

[28] Heather J Goldsby et al. "Goal-based modeling of dynamically adaptive system requirements". In: *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*. IEEE. 2008, pp. 36–45.

[29] Mira Vrbaski et al. "Goal models as run-time entities in context-aware systems". In: *Proceedings of the 7th Workshop on Models@ run. time*. ACM. 2012, pp. 3–8.

[30] Brice Morin et al. "Models@ run. time to support dynamic adaptation". In: *Computer* 42.10 (2009).

[31] Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio. "Runtime monitoring of functional component changes with behavior models". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 152–166.

[32] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. "Mocas: A state-based component model for self-adaptation". In: *Self-Adaptive and Self-Organizing Systems, 2009. SASO'09. Third IEEE International Conference on*. IEEE. 2009, pp. 206–215.

[33] Thomas Vogel and Holger Giese. "Model-driven engineering of self-adaptive software with eurema". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8.4 (2014), p. 18.

[34] Brice Morin et al. "An aspect-oriented and model-driven approach for managing dynamic variability". In: *international conference on Model Driven Engineering Languages and Systems*. Springer. 2008, pp. 782–796.

[35] Daniel Menasce, Hassan Gomaa, Joao Sousa, et al. "Sassy: A framework for self-architecting service-oriented systems". In: *IEEE software* 28.6 (2011), pp. 78–85.

[36] Nelly Bencomo and Gordon Blair. "Using architecture models to support the generation and operation of component-based adaptive systems". In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 183–200.

[37] David Garlan, Robert T Monroe, and David Wile. "Acme: Architectural description of component-based systems". In: *Foundations of component-based systems* 68 (2000), pp. 47–68.

[38] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. "A graph based architectural (re) configuration language". In: *ACM SIGSOFT Software Engineering Notes* 26.5 (2001), pp. 21–32.

[39] Jim Dowling and Vinny Cahill. "The k-component architecture meta-model for self-adaptive software". In: *International Conference on Metalevel Architectures and Reflection*. Springer. 2001, pp. 81–88.

[40] Pete Sawyer et al. "Requirements-aware systems: A research agenda for re for self-adaptive systems". In: *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE. 2010, pp. 95–103.

[41] Éric Tanter et al. "Partial behavioral reflection: Spatial and temporal selection of reification". In: *ACM SIGPLAN Notices* 38.11 (2003), pp. 27–46.

[42] Thais Batista, Ackbar Joolia, and Geoff Coulson. "Managing dynamic reconfiguration in component-based systems". In: *European workshop on software architecture*. Springer. 2005, pp. 1–17.

[43] Gordon S Blair et al. "Reflection, self-awareness and self-healing in OpenORB". In: *Proceedings of the first workshop on Self-healing systems*. ACM. 2002, pp. 9–14.

[44] Robert T Monroe. *Capturing Software Architecture Design Expertise with Armani Version 1.0*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1998.

[45] Mehdi Amoui et al. "Adaptive action selection in autonomic software using reinforcement learning". In: *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*. IEEE. 2008, pp. 175–181.

[46] Gerald Tesauro. "Reinforcement learning in autonomic computing: A manifesto and case studies". In: *IEEE Internet Computing* 11.1 (2007).

[47] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. "A control-based framework for self-managing distributed computing systems". In: *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM. 2004, pp. 3–7.

[48] Werner Brockmann, Nils Rosemann, and Erik Maehle. "A framework for controlled self-optimisation in modular system architectures". In: *Organic Computing—A Paradigm Shift for Complex Systems*. Springer, 2011, pp. 281–294.

[49] Holger Prothmann et al. "Organic control of traffic lights". In: *International Conference on Autonomic and Trusted Computing*. Springer. 2008, pp. 219–233.

[50] Dominik Fisch, Edgar Kalkowski, and Bernhard Sick. "Collaborative learning by knowledge exchange". In: *Organic Computing—A Paradigm Shift for Complex Systems*. Springer, 2011, pp. 267–280.

[51] Jim Dowling and Vinny Cahill. "Self-managed decentralised systems using K-components and collaborative reinforcement learning". In: *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM. 2004, pp. 39–43.

[52] William E Walsh et al. "Utility functions in autonomic systems". In: *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE. 2004, pp. 70–77.

[53] Craig Boutilier et al. "Towards cooperative negotiation for decentralized resource allocation in autonomic computing systems". In: *IJCAI*. 2003, pp. 1458–1459.

[54] Gerald Tesauro et al. "A multi-agent systems approach to autonomic computing". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society. 2004, pp. 464–471.

[55] Vahe Poladian et al. "Dynamic configuration of resource-aware services". In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE. 2004, pp. 604–613.

[56] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated multi-objective control for self-adaptive software design". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 13–24.

[57] Stepan Shevtsov and Danny Weyns. "Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 229–241.

[58] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated design of self-adaptive software with control-theoretical formal guarantees". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 299–310.

[59] Viraj Bhat et al. "Enabling self-managing applications using model-based online control strategies". In: *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*. IEEE. 2006, pp. 15–24.

[60] Gregoris Tziallas and Babis Theodoulidis. "A controller synthesis algorithm for building self-adaptive software". In: *Information and Software Technology* 46.11 (2004), pp. 719–727.

[61] Gabor Karsai et al. "An approach to self-adaptive software based on supervisory control". In: *International Workshop on Self-Adaptive Software*. Springer. 2001, pp. 24–38.

[62] Yuriy Brun et al. "Engineering self-adaptive systems through feedback loops". In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70.

[63] Joseph L Hellerstein et al. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[64] Mary Shaw. "Beyond objects: A software design paradigm based on process control". In: *ACM SIGSOFT Software Engineering Notes* 20.1 (1995), pp. 27–38.

[65] Tarek Abdelzaher et al. "Introduction to control theory and its application to computing systems". In: *Performance Modeling and Engineering*. Springer, 2008, pp. 185–215.

[66] Antonio Filieri et al. "Control strategies for self-adaptive software systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11.4 (2017), p. 24.

[67] Tharindu Patikirikorala et al. "A systematic survey on the design of self-adaptive software systems using control engineering approaches". In: *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*. IEEE. 2012, pp. 33–42.

[68] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia Pearson Education Limited, 2016.

[69] Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[70] Lawrence Davis. "Handbook of genetic algorithms". In: (1991).

[71] Jeremy S Bradbury et al. "A survey of self-management in dynamic software architecture specifications". In: *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM. 2004, pp. 28–33.

[72] Frank D Macías-Escrivá et al. "Self-adaptive systems: A survey of current approaches, research challenges and applications". In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279.

[73] Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai. "Archface: a contract place where architectural design and code meet together". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM. 2010, pp. 75–84.

[74] Abdelkrim Amirat and Mourad Oussalah. "C3: A metamodel for architecture description language based on first-order connector types". In: *11th International Conference on Enterprise Information Systems (ICEIS 2009)*. 2009, pp. 76–81.

[75] Adel Smeda, Adel Alti, and Abbdellah Boukerram. "An environment for describing software systems". In: *WSEAS Transactions on Computers* 8.9 (2009), pp. 1610–1619.

[76] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. "Montiarc-architectural modeling of interactive distributed and cyber-physical systems". In: *arXiv preprint arXiv:1409.6578* (2014).

[77] Robert Allen, Remi Douence, and David Garlan. "Specifying and analyzing dynamic software architectures". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 1998, pp. 21–37.

[78] Nenad Medvidovic and Richard N Taylor. "A classification and comparison framework for software architecture description languages". In: *IEEE Transactions on software engineering* 26.1 (2000), pp. 70–93.

[79] Arvid Butting et al. "A classification of dynamic reconfiguration in component and connector architecture description languages". In: *4st International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp) 2017 Workshop Pre-proceedings*. 2017, p. 13.

[80] Yang Lingling and Zhao Wei. "An Overview of Software Architecture Description Language and Evaluation Method". In: *Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering*. Springer. 2013, pp. 895–901.

[81] Frank Budinsky et al. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[82] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Systems in BIP". In: *SEFM'06 Proceedings*. IEEE Computer Society Press, 2006, pp. 3–12.

[83] Ananda Basu et al. "Rigorous Component-Based System Design Using the BIP Framework". In: *IEEE Software* 28.3 (2011), pp. 41–48.

[84] Marius Bozga et al. "Modeling dynamic architectures using Dy-BIP". In: *International Conference on Software Composition*. Springer. 2012, pp. 1–16.

[85] Carl Bergenhem. "Approaches for facilities layer protocols for platooning". In: *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*. IEEE. 2015, pp. 1989–1994.

[86] Shervin Nouyan et al. "Teamwork in self-organized robot colonies". In: *IEEE Transactions on Evolutionary Computation* 13.4 (2009), pp. 695–711.

[87] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[88] Danny Weyns et al. "A survey of formal methods in self-adaptive systems". In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM. 2012, pp. 67–79.

[89] Gabriel Tamura et al. "Towards practical runtime verification and validation of self-adaptive software systems". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 108–132.

[90] Saddek Bensalem et al. “Compositional verification for component-based systems and application”. In: *IET software* 4.3 (2010), pp. 181–193.

[91] Anastasia Mavridou et al. “Configuration logics: Modeling architecture styles”. In: *Journal of Logical and Algebraic Methods in Programming* 86.1 (2017), pp. 2–29.

ABSTRACT

Modeling Self-configuration in Architecture-based Self-adaptive systems

Modern systems are pressured to adapt in response to their constantly changing environment to remain useful. Self-adaptive systems are able to modify at runtime their behavior and/or structure in response to their perception of the environment, the system itself, and their requirements. The focus of this work is on realizing self-configuration, a key and essential property of self-adaptive systems. Self-configuration is the capability of reconfiguring automatically and dynamically in response to changes. This may include installing, integrating, removing and composing/decomposing system elements. This thesis introduces the Dr-BIP framework, an extension of the BIP framework for modeling self-configuring systems that relies on a model-based and component & connector approach to prescribe systems.

A Dr-BIP system model is a runtime model which captures the running system at three different levels of abstraction namely behavior, configuration, and configuration variants. To capture the three levels of abstraction, we introduce motifs as primary structures to prescribe a self-configuring Dr-BIP system. A motif defines a set of components that evolve according to interaction and reconfiguration rules. A system is composed of multiple motifs that possibly share components and evolve together. Interaction rules dictate how components composing the system can interact and reconfiguration rules dictate how the system configuration can evolve over time. Finally, we show that the proposed framework is both minimal and expressive by modeling four different self-configuring systems. Last but not least, we propose a modeling language to codify the framework concepts and provision an interpreter implementation.

RÉSUMÉ

Modélisation de la configuration automatique dans des systèmes

Pour rester utile, les systèmes modernes doivent s'adapter à leur environnement qui ne cessent d'évoluer. Les systèmes auto-adaptatifs peuvent modifier, au moment de l'exécution, leur comportement et / ou leur structure en fonction de leur perception de l'environnement, du système même et de leurs exigences. L'objectif de ce travail est de réaliser l'auto-configuration, une propriété clé et essentielle des systèmes auto-adaptatifs. L'auto-configuration est la capacité de se reconfigurer automatiquement et dynamiquement suite aux changements, tel que l'installation, l'intégration, le retrait et la composition / décomposition d'éléments du système. Cette thèse présente le cadre du Dr-BIP, une extension du plan BIP pour la modélisation des systèmes à configuration automatique qui repose sur une approche basée sur un modèle et sur des composants et des connecteurs pour prescrire des systèmes.

Un modèle de système Dr-BIP est un modèle d'exécution qui capture le système en cours d'exécution à trois niveaux d'abstraction différents, à savoir du comportement, de configuration et des variantes configurations. Pour capturer les trois niveaux d'abstraction, nous introduisons des motifs en tant que structures principales pour prescrire un système Dr-BIP à configuration automatique. Un motif définit un ensemble de composants qui évoluent en fonction de règles d'interaction et de reconfiguration. Un système est composé de plusieurs motifs pouvant éventuellement partager des composants et évoluer ensemble. Les règles d'interaction dictent la manière dont les composants du système peuvent interagir, tandis que les règles de reconfiguration dictent l'évolution de la configuration du système. Enfin, nous montrons que le cadre proposé est à la fois minime et expressif en modélisant quatre systèmes différents à configuration automatique. Finalement, nous proposons un langage de modélisation pour codifier les concepts du cadre et fournir une implémentation d'interprète.