The DReAM framework: a logic-inspired approach to reconfigurable system modeling

PhD in Institutions, Markets and Technologies

Curriculum in Computer Science

XXX Cycle

By

Alessandro Maggi

2020

The dissertation of Alessandro Maggi is approved.

Program Coordinator: Rocco De Nicola, IMT School for Advanced Studies Lucca

Supervisor: Prof. Rocco De Nicola, IMT School for Advanced Studies Lucca

Supervisor: Prof. Joseph Sifakis, Univ. Grenoble Alpes

The dissertation of Alessandro Maggi has been reviewed by:

Prof. Martin Wirsing, Ludwig-Maximilians-Universität München

Prof. Roberto Bruni, University of Pisa

In beloved memory of my grandmother,
Annunziata Mastracca (June 3, 1921 - November 20, 2016)

Contents

Acknowledgements **x**

Vita and Publications **xiii**

Abstract **xv**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Problem statement 3
- 1.3 Main contributions 5
- 1.4 Outline 7
- 1.5 Origin of the material 8

2 Preliminaries **9**
- 2.1 Approaches to software systems modeling 10
  - 2.1.1 Process Description Languages 11
  - 2.1.2 Architecture Description Languages 12
- 2.2 Approaches to components coordination 15
  - 2.2.1 Exogenous coordination paradigms 15
  - 2.2.2 Endogenous coordination paradigms 16
- 2.3 BIP-based Formalisms 17
  - 2.3.1 BIP: Behavior, Interaction, Priority 17
  - 2.3.2 Dynamic BIP 22
  - 2.3.3 Dynamic Reconfigurable BIP 25

3 The L-DReAM framework **29**
3.1 PIL-based systems 30
3.1.1 Propositional Interaction Logic (PIL) 30
3.1.2 Interacting Components 31
3.1.3 Systems of components 32
3.1.4 Disjunctive and Conjunctive styles in PIL 34
3.2 L-DReAM Syntax and Semantics 36
3.2.1 Static systems with PILOps 37
3.2.2 Disjunctive and Conjunctive styles in PILOps . . . 50
3.2.3 Parametric architectures and dynamic systems . . . 55
3.3 Encoding other formalisms 64

4 The DReAM framework **70**
4.1 Structuring architectures 72
4.1.1 Component Types and component Instances 72
4.1.2 Motif modeling 76
4.2 The DReAM coordination language 80
4.2.1 Declaration expansion for coordination terms . . . 81
4.2.2 Reconfiguration operations 83
4.2.3 Disjunctive and Conjunctive styles in DReAM . . . 88
4.2.4 Operational semantics 91
4.3 Example systems 92
4.3.1 Coordinating flocks of interacting robots 92
4.3.2 Coordinating flocks of robots with stigmergy 95
4.3.3 Reconfigurable ring 96
4.3.4 Simple platooning protocol for automated highways 100

5 Executable implementation **107**
5.1 The jDReAM core architecture 108
5.2 The jDReAM extended architecture 115
5.3 Use cases in practice 119
5.3.1 Coordinating flocks of robots 119
5.3.2 Reconfigurable ring 122
5.3.3 Simple platooning protocol 124

6 Concluding considerations **129**
6.1 Summing up 129
6.2 Related works 132
6.3 Future work 134

A Proofs **136**
A.1 Disjunctive to conjunctive transformation in PIL 136

B jDReAM code examples **139**
B.1 Coordinating flocks of interacting robots 139
B.2 Simple platooning protocol for automated highways 142

Acknowledgements

The memory of the first time I walked through the walls of Lucca knowing I was there to stay is still vividly impressed in my mind. It has been quite a long journey from that day, full of twists and turns as any meaningful road in life truly worth traveling along is. Seeing it through has been a challenging feat, one that would have not been possible without the contribution and support of many people.

A major thank you goes to the SysMA research unit of IMT for having me and, more specifically, to its director Prof. Rocco De Nicola, who introduced me to the field of formal methods and process calculi guiding me through the PhD with patience, kindness, and his signature constructive optimism. I owe him a lot both professionally and personally for his continuous support and understanding.

Another major thank you goes to Prof. Joseph Sifakis, who I must credit first and foremost for his inventive ideas that sparked my interest and shaped all the work that led me to this accomplishment. To him goes all my appreciation for his passionate support, priceless insights, and frank exchanges we had.

I am likewise grateful to the attention my external referees, Prof. Roberto Bruni and Prof. Martin Wirsing, have put into my work, pushing me to keep improving it further.

Another thank you goes to Dr. Marinella Petrocchi and Prof. Francesco Tiezzi, who guided me in the first steps of the PhD with a fruitful and stimulating collaboration which taught me

many important lessons on scientific research.

The most significant turning point in this path has undoubtedly been my acceptance of a position at the Bank of Italy halfway-through the PhD. Despite the challenge of continuing my research activity alongside the commitment of a full-time job, I managed to overcome it also thanks to the support of many colleagues in the Bank.

In particular, I would like to thank the former head of the Supervision Inspectorate Dr. Carmelo Lattuca and the head of the On-site Inspection Planning Division Dr. Paolo Bernardini for authorizing an extended leave from work which allowed me to finish my study period at the School.

I would also like to thank all the members of the Applied Research Team of the General Directorate for Information Technology, where I currently work as a research staff member, and specifically the head of the unit Dr. Marco Benedetti and his deputy Dr. Luigi Bellomarini for their understanding and open support to my external research activities.

No journey is truly complete without travel companions. I am truly grateful I had the opportunity to meet so many along the way, I want to take the time to thank the ones who are dearer to me one by one in no particular order.

Hanin, my dear coffee-pal, for all the coffees, the quality movie and TV-series recommendations, and especially the meaningful and inspirational conversations we had; Paolo, for sharing many priceless moments, both fun and emotional, as well as one of the most peculiar cooking sessions involving a chicken and a can of beer; Luca, for all the funny moments and shower-related puns, as well as for being the best roommate one could hope for; Valentina, for her ability of combining rock-solid determination with contagious glee, and for being the best part-

ner in interdisciplinary projects and team-based card games; Valerio, for his self-irony, his infectious laughter, and his unbounded passion for lemons; Vitaly, for his ever-so-slightly decipherable poker-face and his big heart; Anita, for her enthusiasm, kindness, and for being always there to help others; Claudia, for the shared immeasurable enthusiasm for tabletop- and video-games.

Most importantly, to all of them, thank you for your friendship and for the time we spent together.

Next, I wish to thank my closest friends, Francesco, Germano and Valerio, who have always been there for me even when we were far apart and had less opportunities to hang out.

Many thanks to the brilliant "team members" Fabio and Saimir, always ready to pitch new ideas and arrange epic reunions across space and time ever since we met at the university.

Thank you also to the eclectic duo from my past life as junior physicist, Riccardo and Valerio, with whom I will always remember some of the most peculiar moments of our life spent in a laboratory.

I also owe many thanks to my parents and my aunt Nuccia for their unconditional support and encouragement throughout these years.

Last, but definitely not least, I thank from the bottom of my heart my soulmate and wife Michela, who has always been by my side encouraging me to do my best and to follow my dreams, giving me strength throughout this journey.

Vita

March 22, 1985	Born, Rome, Italy
2008	Bachelor Degree in Physics Final mark: 106/110 Sapienza University of Rome, Rome, Italy
2008-2010	Collaborator journalist Soura Magazine, Unexplored Publishing LLC, Dubai, United Arab Emirates
2009-2015	Collaborator journalist Notebook Italia, Trani, Italy
2010	Bachelor Degree in Computer Engineering Final mark: 110/110 Roma Tre, Rome, Italy
2013	Master Degree in Computer Engineering Final mark: 110/110 cum laude Sapienza University of Rome, Rome, Italy
2013-2015	Photographer RPM Proget, Rome, Italy
2014	Business Technologist Capgemini Italia, Rome, Italy
since 2014	PhD Candidate in Computer Science IMT School for Advanced Studies Lucca, Lucca, Italy
2016-2019	Administrative assistant Supervision Inspectorate, Bank of Italy, Rome, Italy
since 2019	ICT research staff member Applied Research Team, Bank of Italy, Rome, Italy

Publications

1. R. De Nicola, A. Maggi, M. Petrocchi, A. Spognardi, and F. Tiezzi. "Twitlang(er): Interactions Modeling Language (and Interpreter) for Twitter". In: *Proceedings of the 13th International Conference of Software Engineering and Formal Methods*. Springer, 2015.
2. A. Maggi, M. Petrocchi, A. Spognardi, and F. Tiezzi. "A language-based approach to modelling and analysis of Twitter interactions". In: *Journal of Logical and Algebraic Methods in Programming*, vol. 87, 2017.
3. [43] R. De Nicola, A. Maggi, and J. Sifakis. "DReAM: dynamic reconfigurable architecture modeling". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018.
4. [37] A. Maggi, R. De Nicola, and J. Sifakis. "A Logic-Inspired Approach to Reconfigurable System Modelling". In: *From Reactive Systems to Cyber-Physical Systems*. Springer, 2019.
5. [18] R. De Nicola, A. Maggi, and J. Sifakis. "The DReAM framework for dynamic reconfigurable architecture modelling: theory and applications". In: *International Journal on Software Tools for Technology Transfer*, 2020.

Abstract

Modern systems evolve in unpredictable environments and have to continuously adapt their behavior to changing conditions. The DReAM (Dynamic Reconfigurable Architecture Modeling) framework has been designed to address these requirements by offering the tools for modeling reconfigurable dynamic systems effectively.

At its core, the framework allows component-based architecture design leveraging a rule-based language, inspired from Interaction Logic. The expressiveness of the language allows us to define the behavior of both components and components aggregates encompassing all aspects of dynamicity, including parametric multi-modal coordination of components and reconfiguration of their structure and population.

DReAM allows the description of both endogenous/modular and exogenous/centralized coordination styles and sound transformations from one style to the other, while adopting a familiar and intuitive syntax. To better model dynamic mobile systems, the framework is further extended with two structuring concepts: *motifs* - independent dynamic architectures coordinating components assigned to them - and *maps* - graph-like data structures modeling the topology of the environment and parametrizing coordination between components.

The jDReAM Java project has been developed to provide an execution engine with an associated library of classes and methods that support system specifications conforming to the DReAM syntax. It allows to develop runnable systems combining the expressiveness of the rule-based notation with the flexibility of this widespread programming language

Chapter 1

Introduction

1.1 Motivation

Depending on the perspective from which we examine a software system, we could consider it a procedure made of instructions that a machine can understand and execute, a mathematical function that maps an input to an output, a circuit of elements in which data and control signals flow through ports appropriately wired, and so on. Different aspects and properties emerge by adopting one perspective or the other. The same applies if we shift our attention on the approaches to design such systems. Depending on the level of abstraction that we choose to adopt for each element that contributes to the definition of what a system does, we have to account for a trade-off between expressive power, practical viability in terms of actual instantiation and implementation, ease of human understanding, and verifiability of desired properties (such as correctness and liveness).

Modern programming languages have reached a level of diffusion and maturity that arguably grants sufficient readability of code in addition to high expressiveness and efficient execution. On the other hand, the challenges posed by the limits of verifiability of their broad capabilities and rich data types limits the kind of guarantees that we can obtain, so errors and unexpected behaviors are checked mostly during testing

and get patched even after the program is released. This is a standard practice in the industry, and while there is no reason to require that every single line of code of every software has to be error-free from the day of release, there are two exceptions that require additional guarantees:

1. *critical systems* whose correctness has direct implications for human health and safety: potential faults have to be tracked and fixed before they manifest during execution, as the consequences can be major and irreversible;

2. *open systems* executing in an unpredictable environment and coordinating with third-party agents not known *a priori*: correctness checks of system code could be not enough as unexpected, erroneous behaviors can emerge from the interaction with external entities.

The ever increasing complexity of modern software systems has changed the perspective of software designers who now have to consider new classes of them, consisting of a large number of interacting components and featuring complex interactions. These systems are usually distributed, heterogeneous, decentralized and interdependent, and are operating in an unpredictable environment. Such level of complexity calls for new approaches to software design taking advantage of decomposition, indirection and adaptation, as the lack of rigorous methodologies and formally grounded modeling frameworks increases the risks of undesirable interference and unpredictable outcomes. To cope with these issues, software needs to be developed in such a way that systems can continuously adapt to internal changes and to changes in their operating environment.

The notion of "adaptation" directly ties in with that of "reconfiguration" capability, which allows to capture the dynamic aspects of modern software already at design-time. Integrating these capabilities in the design process is even more challenging for frameworks that aim at modeling architectures, as they require to support both parametric systems instantiations and dynamic reconfigurations of their inner structure and of their coordination patterns. Software architectures are crucial in software development as they represent the fundamental structures upon which

individual systems are built. Effective architecture modeling supports efficient and correct development by encouraging modularity, separation of concerns, and rationalization of recurring scenarios into "patterns" and "styles". Despite the challenges, these benefits are all the more needed for the classes of systems that we described. Architecture modeling languages should be expressive enough to support these features while offering an intuitive syntax and clear design methodologies in order to foster their practical use.

1.2 Problem statement

One of the main challenges that formalized approaches to complex systems specification have to face is how to capture their dynamic aspects and how to integrate them in a rigorous and controlled way in the design process. Dynamism, indeed, plays a crucial role in these modern systems and can be guaranteed by exploiting the expressive power offered by the three following features:

1. the parametric description of interactions between instances of components for a given system configuration;

2. the reconfiguration involving creation/deletion of components and management of their interaction according to a given architectural style;

3. the migration of components between predefined architectural styles.

The first feature implies the ability of describing the coordination of systems that are parametric with respect to the number of instances of types of components; examples of such systems are Producer-Consumer systems with m producers and n consumers or Ring systems consisting of n identical interconnected components.

The second feature is related to the ability of reconfiguring systems by creating or deleting components and managing their interactions taking into account the dynamically changing conditions. In the case of a reconfigurable ring this would require having the possibility of removing

a component which self-detects a failure and of adding it back after recovery. Added components are subject to specific interaction rules according to their type and their position in the system. This is especially true for mobile components which are subject to dynamic interaction rules depending on the state of their neighborhood.

The third aspect is related to the vision of "fluid architectures" [24] or "fluid software" [55]. The underlying idea is that applications and "smart" objects live in an environment which corresponds to an architectural style that defines how these entities behave and interact. Systems' dynamicity is modeled by allowing such entities to migrate among different environments; this dynamic migration allows a disciplined and easy-to-implement management of dynamically changing coordination rules. For instance, self-organizing systems may adopt different architectural styles to adapt their behavior in order to guarantee global properties.

The different approaches to architectural modeling and the new trends and needs are reviewed in detailed surveys such as [13, 16, 38, 39, 45]. Here, we consider two criteria for the classification of existing approaches: *exogenous vs. endogenous* and *conjunctive vs. disjunctive* modeling.

Exogenous modeling assumes that components are architecture-agnostic and guarantee a strict separation between a component behavior and its coordination. Coordination is specified globally by rules applied to sets of components. The rules involve synchronization of events between components and associated data transfer. This approach is adopted by Architecture Description Languages (ADL) [38]. It has the advantage of providing a global view of the coordination mechanisms and their properties.

Endogenous modeling requires adding explicit coordination primitives in the code describing components behavior. Components are composed through their interfaces, which are used to expose their coordination capabilities. An advantage of endogenous coordination is that it does not require programmers to explicitly build a global coordination model. However, validating a coordination mechanism and studying its properties becomes much harder without such a model.

Conjunctive modeling uses logic to express coordination constraints

between components. In particular, it allows modular description of compound systems as one can associate with each component its coordination constraints. The global coordination of a system can then be obtained as the conjunction of individual constraints of its constituent components.

Disjunctive modeling consists in explicitly specifying system coordination as the union of executable coordination mechanisms such as semaphores, function call and connectors.

Merits and limitations of the two approaches are well understood. Conjunctive modeling allows abstraction and modular description but it is exposed to the risk of inconsistency in case there is no architecture satisfying the specification. Conversely, disjunctive modeling by construction does not expose system designers to this risk, but requires the definition of monolithic connectors that usually have strong dependencies with other parts of the system and thus have limited reusability.

Many approaches for software architecture modeling have been proposed in literature with varying degrees of formalization, but to the best of our knowledge none of them provides a generalized foundation capable of capturing all the listed modeling criteria while also supporting dynamic reconfiguration and architectural style changes at runtime.

1.3 Main contributions

This thesis details two closely connected formal frameworks for the specification of software architectures which natively support dynamism and advanced reconfiguration capabilities. Both offer rigorous methodologies for architecture design by adopting a logic-based modeling language that is expressive and powerful enough to support different approaches to coordination and all the key features required to capture dynamicity. As such, the two frameworks do not impose limits to the coordination and computation styles, allowing instead to formalize sound transformations between different design approaches.

L-DReAM - Light Dynamic Reconfigurable Architecture Modeling - is the most "unstructured" of the two frameworks aimed at capturing

all the core aspects of the underling modeling language with minimal overhead.

In L-DReAM a system is a hierarchical structure of components. Each non-atomic component hosts a "pool" of components and defines the coordination rules that regulate the way they interact and evolve. The overall structure of systems can also change as components can leave and join different pools. Components will thus be subject to different coordination rules as they change their position in the hierarchy.

L-DReAM rules include an interaction constraint, modeled as a formula of the Propositional Interaction Logic (PIL) [11], and some operations allowing data transfer as well as more complex reconfigurations of the component's state. Parametric coordination between classes of components is achieved through the introduction of the concepts of *component types* - blueprints for actual components - and *component instances* created from specific types. Their coordination is characterized by rules in a first order extension to PIL with quantification over instance variables of a given type.

DReAM - Dynamic Reconfigurable Architecture Modeling - can be thought as a specialization of L-DReAM to model systems that do not require unbounded, free-form structures of components, but instead adopts a hierarchy that adequately fits concrete systems and integrates additional parametrization structures that make the design of dynamic and mobile architectures more intuitive and efficient.

In DReAM, a system consists of instances of types of components organized in hierarchies of "motifs". Component instances can migrate between motifs depending on specific conditions. Thus, a given type of component can be subject to different rules when it is in a "ring" motif or in a "pipeline" one. Motifs themselves are typed, and each motif instance can migrate from one to another, dynamically changing their hierarchy and thus potentially affecting all the underlying components. Using motifs allows a natural description of self-organizing systems.

Coordination rules in a motif are defined according to the same core language used by L-DReAM. To enhance the expressiveness of the differ-

ent kinds of dynamism, each motif is also equipped with a "map", which is a graph defining the topology of the interactions in the motif. Components are associated with locations of the map via an addressing function @ which defines the position $@(c)$ in the map of any component instance c associated with the motif. Additionally, each location is equipped with a local memory that can be accessed by components and used as a shared memory. Maps are also very useful to express mobility, in which case the connectivity relation of the map represents possible moves of components. Finally, DReAM allows us to modify maps by adding or removing locations and edges, as well as to dynamically create and delete component instances.

1.4 Outline

The rest of the thesis is organized as follows.

Chapter 2 gives an overview of the "setting" in which the problem of defining software intensive systems has been faced so far, focusing on how different approaches handle the architectural aspects of system definitions, how they capture dynamic characteristics of the systems at runtime, and to what extent they support adaptation through reconfiguration of their structure and behavior. This chapter will also present some of the main formalisms that have inspired the development of the two frameworks described in this thesis.

Chapter 3 introduces L-DReAM, a "light" variant of the DReAM framework that shares its approach on coordination of components but retains a simpler syntax and a looser structure, making it very flexible and more suitable to study fundamental properties of the underlying language. The presentation of the framework is done incrementally, starting from a simple static setting where components interact according to formulas of the Propositional Interaction Logic (PIL), and proceeding upwards by first adding data transfer between components and then reconfiguration capabilities.

Chapter 4 presents the fully-structured DReAM framework detailing its key concepts and features, both in terms of their role in addressing the

requirements for dynamic and reconfigurable architecture modeling as well as in the way they tie in with fundamental L-DReAM concepts. To highlight the distinguishing characteristics of DReAM with respect to its light variant and other formalisms, we discuss few specific use cases that take full advantage of the capabilities of the framework.

Chapter 5 describes the prototype Java-based modeling and execution framework jDReAM with a detailed overview of its architecture. The core components of jDReAM implement the abstract syntax of L-DReAM and its operational semantics, while a dedicated library extends the API in order to support DReAM specifications natively.

The concluding Chapter summarizes the main contributions and results, briefly highlights similarities and differences between related works and the presented frameworks, and discusses directions for future work both from a theoretical and practical standpoint.

1.5 Origin of the material

The contents of this Thesis refer to and expand upon the following publications:

[43] R. De Nicola, A. Maggi, and J. Sifakis. "DReAM: dynamic reconfigurable architecture modeling". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018.

[37] A. Maggi, R. De Nicola, and J. Sifakis. "A Logic-Inspired Approach to Reconfigurable System Modelling". In: *From Reactive Systems to Cyber-Physical Systems*. Springer, 2019.

[18] R. De Nicola, A. Maggi, and J. Sifakis. "The DReAM framework for dynamic reconfigurable architecture modelling: theory and applications". In: *International Journal on Software Tools for Technology Transfer*, 2020.

Chapter 2

Preliminaries

Software systems have reached a level of complexity that calls for more rigorous approaches to software design. When applied to classes of systems that globally consist of countless interacting components and feature complex interaction patterns, the lack of rigorous methodologies and formally grounded frameworks for modeling them increases the risks of undesirable interference and unpredictable outcomes. These risks are amplified by the fact that these systems are usually distributed, heterogeneous, interdependent, and are operating in unpredictable environments. To cope with these issues, software needs to be developed in order to continuously adapt to internal changes and to external ones happening in the environment.

Correct design of such systems, even those traditionally not classified as "critical", is becoming more and more important as they become increasingly interconnected and the fault of few can cause ripple effects on many others. This challenge has been faced by practitioners using two very distinct approaches:

1. a "production-centric" approach revolving mainly on organizational frameworks for the software life-cycle management that leverage best practices to guide software design and development;

2. a "formal" approach providing sound mathematical models of com-

putation that can be used to specify software systems and verify qualitative and quantitative properties theoretically.

Discussing the merits and limitations of the former approach is beyond the scope of this Thesis, which will focus on the latter instead.

This Chapter is organized as follows.

Section 2.1 presents an overview of two different approaches to software modeling, namely *Process Description Languages* and *Architecture Description Languages*.

Section 2.2 shifts the perspective to coordination paradigms for component-based software modeling, comparing the *exogenous coordination* approach with the *endogenous* one.

Section 2.3 focuses on the BIP framework and some of the main representatives of its derived formalisms showing how they relate with the taxonomy discussed in Section 2.2. This will also provide the basis upon which the frameworks discussed in this Thesis have been devised.

2.1 Approaches to software systems modeling

To formalize a system specification there are many approaches that can be adopted which depend on the perspective that we assume. We could focus on the description of how individual elements of a system behave by defining precise rules that allow to reconstruct the exact execution trajectory down to its sequence of states. However, if we are more interested in capturing dependency relationships between components that constitute a system and how they interact with one another, we might adopt instead an approach that highlights the overall structure of the system and treats other aspects with coarser approximation. These are just two examples in a wide range of different dimensions in which it is possible to organize the many formalisms and frameworks that allow to write software-intensive system specifications. Among these, two broad categories relating to the two orthogonal perspectives that we have just introduced are *Process Description Languages* and *Architecture Description Languages*.

2.1.1 Process Description Languages

Process Description Languages (PDLs) are backed by a rich theory developed in the last three decades that formalizes the semantics of concurrent programming allowing to understand various functional and non-functional aspects of concurrent, distributed and mobile systems.

As of now there exist a large number of instances of PDLs, but overall they share a common ground: they can be generally described as *action-based formalisms* relying on behavioral operators that support *compositionality* and *abstraction*, where the meaning of process terms is formally defined via structural operational semantics rules that induce a state transition graph over the process term.

Compositionality and abstraction are key concepts in the formalism as they enable the definition of *behavioral equivalences*. Such equivalences permit manipulations at syntax and semantics level that allow to focus on specific aspects of a system and to fix the necessary granularity of the specification by possibly abstracting from unnecessary details. By exploiting them it is possible to verify whether a system behavior conforms to a given *specification*, i.e., a reference model describing the expected abstract behavior of an application. This approach is generally called *equivalence checking*.

PDLs, when equipped with an operational semantics, do support also another kind of verification technique known as *model checking*. The goal of this approach is to verify whether a given system satisfies some properties, often expressed via an appropriate logic. The fundamental difference between model checking and equivalence checking lies in the fact that properties need not be a full specification of the intended behavior, but they might deal with specific characteristics of the system (e.g. deadlock freedom).

Notable representatives of PDLs well recognized in scientific literature are, among others, CSP (Communicating Sequential Processes) [28, 29], CCS (Calculus of Communicating Systems) [40] and π-calculus [41].

The expressive power and flexibility of this class of formalisms lent themselves to extensions that go beyond the initial scope of modeling

functional system behavior. Few examples of them allowing to capture quantitative features are *timed* and *probabilistic* process calculi [3, 26, 44].

2.1.2 Architecture Description Languages

Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software-intensive system. With the term *architecture* we refer to the description of the components that comprise a system, the behavioral specifications of those components, and the patterns and mechanisms for their interactions. Note that a single system is usually composed of more than one type of component: modules, tasks, functions, etc. An architecture can choose the type of components most appropriate or informative, or it can include multiple views of the same system, each illustrating different components.

As software architectures became a dominating theme in large system development and acquisition, methods for unambiguously specifying them have become more and more important. The main notions were already identified in the early '90s [25, 57]: components abstracting computations and data repositories, connectors abstracting the interaction protocols used among components. Since then, a rich research literature has accompanied the development of several ADLs, experimenting with different approaches to architecture specification.

Despite the efforts put into place by the scientific community, rigorous approaches to software architectures have yet to reach mainstream attention within the community of developers. Indeed, as highlighted in [38] and [58], the process of software development in the industry has been heavily influenced by organizational frameworks fostering "agile" methodologies that encourage quick iterations and fast prototyping, but has not yet tied these practices with formalized frameworks that bind architectural descriptions with their implementations. In fact, practitioners are still heavily relying on informal circle-and-line drawings and semi-formal approaches that exhibit very weak support for architecture specification such as UML [32]. This makes traceability between architectural documentation and actual code ambiguous and challenging, diminish-

ing the incentive to keep the former up-to-date with the latter, which in turn discourages a wider adoption of structured approaches addressing architectural aspects of software design like ADLs.

However, agile development is not inherently in contrast with rigorous architecture description approaches. The reasons behind the "resistance" to ADLs adoption are manifold. On one hand the constellation of proposed languages and their different characteristics prevents potential adopters to get a clear picture of what each one has to offer and how using them could be beneficial in the scope of software development. Furthermore surveys on the main acknowledged ADLs [17, 22, 47] have revealed the presence of "gaps" in the formal semantics associated to some of the languages, thus making early analysis of system architectures in such instances difficult or even impossible. Paired with what seems to be one of the biggest obstacles in the adoption - i.e. the complexity of the languages and the absence of an industrial standard [38] - this results in spotty usage of the more sophisticated languages for architectures specification.

Commonalities and discriminators among ADLs

Early architecture description languages such as ArTek [56], CODE [42], COMMUNITY [21], Darwin [36], Demeter [48], Modechart [34], PSDL/-CAPS [8], Rapide [35], Resolve [19], UniCon [54] and Wright [4], provided an already rich and diversified collection of approaches to the problem before the turn of the century.

Comparative studies [17] showed how each language agreed on a number of common aspects that contributed to an empirical notion of what constitutes an ADL, and conversely exhibited differentiating factors that traced back to the area of expertise of the groups behind their development.

Among the common characteristics shared by most of these ADLs it is worth mentioning:

– presence of a *formally-defined graphical and textual syntax* (usually accompanied with the respective semantics);

- capability to handle data flow and control flow as *interconnection mechanisms*;
- support for *creation*, *validation* and *refinement* of architectures;
- ability to represent *hierarchical levels of detail* and to handle *multiple instantiations of templates*;
- absence of significant support for capturing design rationale and/or history.

Among the discriminant aspects which differentiated these early ADLs the following stand out:

- ability to handle *real-time constructs* at the architectural level;
- support for specification of particular *architectural styles*;
- ability to allow a user to define *new types of components and connectors*, define *new statements* and represent non-architectural information;
- possibility of user-defined definitions of the concept of "consistency";
- ability to handle variability at the level of *architecture instantiation*;
- support to systems *analysis*.

The last point of discrimination deserves additional attention because providing means to perform analysis of architectures is in fact one of the main arguments supporting the usage of ADLs against general-purpose languages like UML.

Indeed while most of the languages supported analysis capabilities, in their initial versions few of them provided actual tools to perform architecture analysis. Among those which offered such tools, the approach to analysis was also very different between one another. For instance, Rapide features a discrete-event simulator based on partially-ordered sets of event behaviors, while Modechart uses a model-checker to verify

whether a system description guarantees, prohibits or is compatible with a given logical assertion.

More recent languages for architecture design - such as AADL [20], ADR [15], Alloy [33], CONNECT [31], COSA [46], PiLar [53], PRISMA [49] and SOFA [52] - proved to offer consistently a broader coverage of features while balancing out complexity by narrowing their scope (e.g., enabling specification of architectural styles but limiting it to their coordination patterns, or offering complete typing of structural concepts and equivalence relations between them but only for static parametric systems). As shown in [47], the diversification process involved mainly the focus on specific capabilities of the language or the support tools, but they can be roughly arranged in two broad categories: the *component-only* approach and the *component-and-connector* one. We will go into more details on them and how they compare to each other in section 2.2.

2.2 Approaches to components coordination

In section 2.1 we provided an overview of two broad categories of approaches to model software-intensive systems. Another way of looking at them is focusing on the way *coordination* between elements of such systems is realized, i.e., how to define which actions synchronize/interleave and under which circumstances. We can single out two main coordination paradigms, namely the exogenous and the endogenous ones.

2.2.1 Exogenous coordination paradigms

The *exogenous* approach to coordination assumes that components are architecture-agnostic, self-contained building blocks that can be combined to form a complex system by "gluing" them together. The glue here is what defines coordination between components, and can be specified in multiple ways, e.g. by means of specific *connectors* that define explicitly which components can interact, under which conditions and to what effect.

This paradigm guarantees a strict separation between a component

behavior and its coordination capabilities with other components. As such, it is conceptually closer to the domain of ADLs.

One of the main advantages of the exogenous approach is that components can be designed individually as they are architecture and coordination agnostic, and thus support code reuse. The architecture defining how components interact and communicate (i.e., the glue) can be designed separately, and to do so only some knowledge of the *interface* of the involved components (i.e., the minimal information that characterizes their endpoints for interaction) is required. An approach that can be considered a representative of this paradigm is REO [5], which indeed focuses purely on the composition of connectors abstracting the inner behavior of components.

This "global" perspective from which coordination between components is defined offers full expressive power to realize any interaction pattern, but for large systems it can make architecture specification cumbersome as it cannot always be developed incrementally.

2.2.2 Endogenous coordination paradigms

The *endogenous* approach requires adding explicit coordination primitives in the code describing components behavior.

Many PDLs can be related to this approach: for example all those where synchronization between components happens by matching compatible actions embedded in their individual behaviors (e.g., the interface parallel operator in CSP). Systems specified with this paradigm can be built incrementally by composing individual components via specific operators such as the parallel composition and the interface parallel.

An advantage of endogenous coordination is that it does not require programmers to explicitly build a global coordination model when writing the specification of a system. The coordination model emerges as the result of the composition of individual behaviors in which the communication actions are embedded. However, validating a coordination mechanism and studying its properties becomes much harder without such a coordination model, requiring specific techniques (such as model

checking for process calculi) that usually can only be applied to subsets of a system, simplified versions of it, or under specific hypotheses.

2.3 BIP-based Formalisms

BIP can be considered a "close relative" of ADLs in which systems are defined as sets of *components* glued together using *connectors*. Despite sharing some structuring concepts and a fundamentally exogenous coordination approach, BIP differentiates from many ADLs in that it offers a fully formalized operational semantics, a mathematical model to represent, combine and analyze interactions, and a comprehensive set of tools for code synthesis, embedding and verification. This makes BIP a viable choice to design and model different classes of systems, but also a robust platform upon which several modeling formalisms have been built - including the frameworks covered by this Thesis.

2.3.1 BIP: Behavior, Interaction, Priority

BIP was developed to model heterogeneous real-time component-based systems. Within the framework, components are obtained as the superposition of three layers that give name to the framework itself: *Behavior*, *Interaction* between transitions of the behavior in the form of a set of *connectors*, and *Priority* to describe the scheduling policies of interactions.

The framework is designed to allow *incremental* construction of systems starting from atomic components (see figure 2) combined through a parametrized binary *composition* operator [27] that acts on each respective layers separately and preserves deadlock-freedom. Parameters are used to define new interactions as well as new priority rules between the composed components.

BIP also supports heterogeneity by providing two mechanisms for structuring interaction: *rendezvous* for strong synchronization, and *broadcast* for weak synchronization.

Atomic components The structured syntax of atomic components is reported in figure 1. Atomic components in BIP are characterized by:

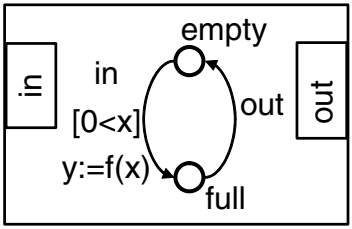
- a set of *ports* $P = \{p_1 \dots p_n\}$, names used for synchronization with other components;
- a set of *control states* $S = \{s_1 \dots s_k\}$, denoting locations at which components wait for synchronization;
- a set of *variables* V used to store data;
- a set of *transitions* modeling atomic computation steps, each one being a tuple of the form (s_1, p, g_p, f_p, s_2) representing a step from control state s_1 to s_2 with guard g_p (Boolean condition on V) and internal computation f_p on V.

It should be noted that guards and statements in the behavior of the component are C expressions and statements, respectively, and that they are restricted with the assumption of atomicity for transitions (e.g. they have no side-effects and their termination is guaranteed).

atom::=
component *component_id*
port *port_id*$^+$
[**data** *type_id data_id*$^+$]
behavior
{**state** *state_id*
{**on** *port_id* [**provided** *guard*]
[**do** *statement*] **to** *state_id*}$^+$}$^+$
end
end

Figure 1: Abstract syntax for atomic components [9]

Figure 2 illustrates an example of an atomic BIP component with two ports (*in* and *out*), two control states (*empty* and *full*), two variables (x and y), and two transitions (*in* and *out*). Notice that the transition *in* from control state *empty* to *full* has a guard and a statement associated to it, requiring that the variable x is greater than 0 to activate and causing y to get assigned the value $f(x)$ as a result.



(a) Graphical representation

component $Reactive$
  port in, out
  data int x, y
  behavior
    state $empty$
      on in **provided** $0 < x$ **do** y:=f(x) **to** $full$
    state $full$
      on out **to** $empty$
  end
end

(b) Description in BIP

Figure 2: An atomic component in BIP (2a) and its formal description (2b) [9]

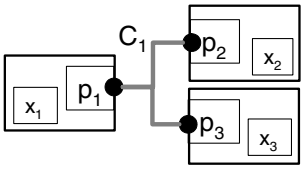
Connectors and interactions Components are built from sets of atomic components with disjoint sets of names for ports, control states, variables and transitions.

A connector γ is a set of ports of atomic components which can be involved in an interaction. Assuming that connectors contain at most one port from each atomic components in the system, an interaction of γ is any non empty subset of this set. Each interaction involving more than one port represents a synchronization between transitions labeled with its ports.

As already mentioned, interactions of a connector γ can express two basic modes of synchronization:

– *strong synchronization* or *rendezvous*: the only feasible interaction of γ is the maximal one (e.g. see figure 3);

– *weak synchronization* or *broadcast*: all the feasible interactions are those containing a particular port which initiates the broadcast (e.g. see figure 4).

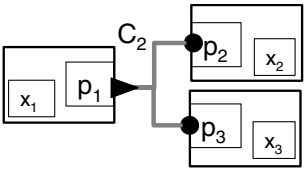


(a) Graphical representation

connector $C_1 = p_1|p_2|p_3$
behavior
on $p_1|p_2|p_3$ **provided** $\neg(x_1 = x_2 = x_3)$
do $x_1, x_2, x_3 := \text{MAX}(x_1, x_2, x_3)$
end

(b) Description in BIP

Figure 3: A connector representing strong synchronization between the components involved in BIP (3a) and its formal description (3b) [9]



(a) Graphical representation

connector $C_2 = p_1|p_2|p_3$
complete = p_1
behavior
on p_1 **do** skip
on $p_1|p_2$ **do** $x_2 := x_1$
on $p_1|p_3$ **do** $x_3 := x_1$
on $p_1|p_2|p_3$ **do** $x_2, x_3 := x_1$
end

(b) Description in BIP

Figure 4: A connector representing weak synchronization between the components involved in BIP (4a) and its formal description (4b) [9]

A connector description includes its set of ports followed by the optional list of its minimal complete interactions and its behavior (figure 5). The latter can be specified by a set of guarded commands associated with feasible interactions like for transitions of atomic components.

interaction ::= *port_id*$^+$
connector::=
connector *conn_id* **=** *port_id*$^+$
[**complete =** interaction$^+$]
[**behavior**
{**on** interaction [**provided** *guard*] [**do** *statement*]}$^+$
end]

Figure 5: Abstract syntax for connectors [9]

Priorities Priorities are used to filter interactions among the feasible ones depending on given conditions (refer to figure 6 for their syntax). Namely, priorities are a set of rules consisting of ordered pairs of interactions associated with a condition on the variables of the components involved in the interactions: when the condition holds and both interactions are enabled, only the one with higher priority is possible.

priority::=
priority *priority_id* [**if** *cond*] interaction $<$ interaction

Figure 6: Syntax for priorities [9]

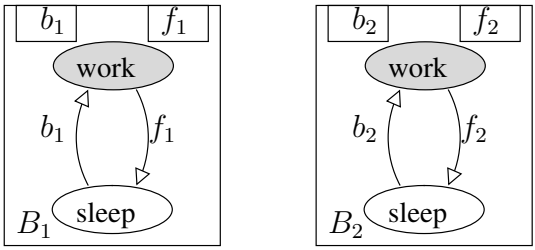
Architectures in BIP

Generally speaking, an architecture can be considered as an operator A that, applied to a set of components $\mathcal{B}$, realizes a composite component $A(\mathcal{B})$ that satisfies a characteristic property Φ.

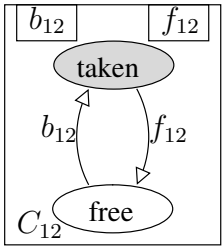
The need of combining multiple architectural solutions over a set of components to achieve some global property by defining an operator "$\oplus$" for *architecture composition* is stressed in [6].

In other words, if we consider two architectures A_1 and A_2 enforcing respectively properties Φ_1 and Φ_2 on a set of components $\mathcal{B}$ - that is $A_1(\mathcal{B}) \models \Phi_1$ and $A_2(\mathcal{B}) \models \Phi_2$ - then it is desirable that the composed architecture $A_1 \oplus A_2$ applied to the same set of components satisfies both properties - i.e. $(A_1 \oplus A_2)(\mathcal{B}) \models \Phi_1 \wedge \Phi_2$.

In this formalization, architectures are solutions to specific coordination problems associated to the relevant properties that must hold. Essentially, an architecture enforces constraints on the possible interactions over the ports of the components involved. For instance, in the example reported in figure 7, the architecture ensuring mutual exclusion between the working components is the one that allows only the interactions between the following pairs of ports: $\{b_1, b_{12}\}, \{b_2, b_{12}\}, \{f_1, f_{12}\}, \{f_2, f_{12}\}$.



(a) components



(b) coordinator

Figure 7: A simple "mutual exclusion" example scenario [6]

The composition operator defined in [6] is associative, commutative and idempotent. The characteristic properties that are considered are safety and liveness.

2.3.2 Dynamic BIP

BIP is quite a mature framework that provides robust features and comprehensive tools to model and develop a variety of systems. However,

like many other ADLs, it lacks effective means to represent *dynamic architectures*. This category of architectures is necessary to model *reconfigurable and adaptive systems*, defined as the composition of components subject to dynamically changing architectural constraints.

One proposed solution developed to support correct and robust modeling of such systems is Dy-BIP [12], a language that can be considered as an extension of BIP natively supporting the design of dynamic architectures.

While Dy-BIP is inspired by BIP, to the best of our knowledge at the current stage it maintains marginal similarities with the original language and needs to be extended in order to be able to model even basic real systems.

Dy-BIP retains the concepts of systems of (atomic) components interacting through sets of ports, but it does not encompass priorities nor connectors.

To realize dynamic architectures, atomic components provide *interaction constraints* for each possible transition and *history variables* that parametrize them keeping track of already executed interactions. Given that each constraint is associated to one port and defines which conditions must hold for it to be involved in an interaction, the coordination paradigm adopted in Dy-BIP can be considered endogenous (i.e., system coordination is the result of the combination of each individual interaction constraint associated to each port). These constraints are expressed using Propositional Interaction Logic (PIL) [11], which is characterized by the following syntax:

$$\Psi ::= \mathbf{true} \mid p \in h \mid p \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \tag{2.1}$$

where $p \in \mathcal{P}$ is a port and $h \in \mathcal{H}$ is an history variable.

To allow the generalization of constraints to parametric system with a finite number of components, PIL is extended to "First-Order Propositional Interaction Logic" (FOIL) with first order quantifiers over *instances*

of *component types*:

$$\Psi ::= \mathbf{true} \mid x.p \in h \mid x.p \mid x = y \mid x = \texttt{self} \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \mid \forall x : T.\Psi\,(x) \tag{2.2}$$

where T is a component type, which represents a set of component instances with the same interface of ports and behavior, and x, y are variables ranging over component instances in the scope of their respective quantifier.

The composition semantics uses a centralized execution engine which gathers current state interaction constraints from all atomic components, then builds and solves the global system of constraints finding the set of maximally satisfying interactions. Finally one of them is selected and executed atomically by all involved components, and the process restarts over the newly reached global state.

To give a glimpse of the language and how it can be used to design software architectures, we will present and comment a simple example from [12].

Example 2.3.1. Let us consider a simple system consisting of two types of components: *Masters* and *Slaves*. Each Master performs two sequential requests to two different Slaves and then performs some computation with them. Figures 8 and 9 show the representation of such types in Dy-BIP.

Each component instance in the system is a transition system sharing the graph representation of its component type. Action names - referred to as *ports* - are associated with *interaction constraints*, which are used to restrict the choice of allowed interactions, and *history variable* updates to keep track of already executed interactions.

Interaction constraints are defined by the syntax of FOIL (2.2), but are written via more convenient abbreviations that express *causal*, *acceptance* and *filtering constraints*. In figure 8, the req port belonging to the transition from s_1 to s_2 presents three different abbreviations:

- **Require** $Slave.get$ is a causal constraint that expresses the requirement for a port get, belonging to a component instance whose type is $Slave$, to be present in the interaction in order for req to participate;

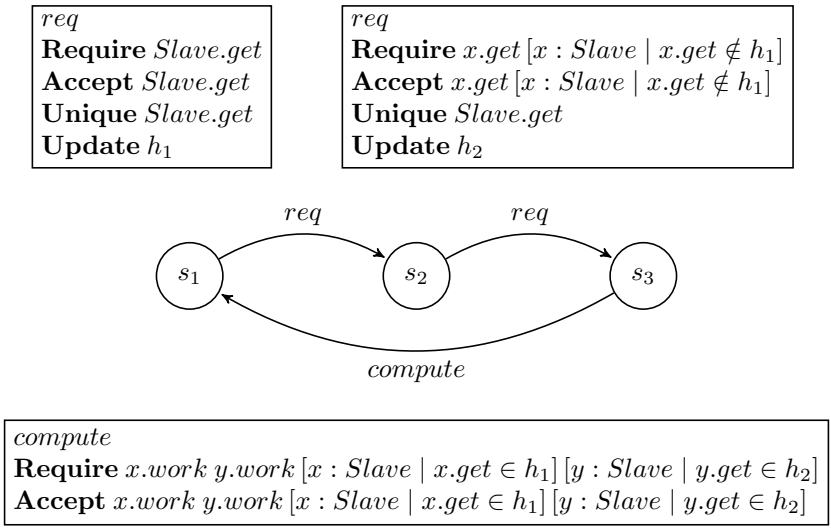


Figure 8: *Master* type

– **Accept** $Slave.get$ is an acceptance constraint that excludes from possible interactions all ports that are not a get of a $Slave$ component type;

– **Unique** $Slave.get$ is a filtering constraint that forbids the participation of req in an interaction with more than one instance of component type $Slave$ with port get.

Richer variants of these abbreviation can involve more than one port/-type combination (e.g. the constraint associated to the $compute$ port of type $Master$ in figure 8) or include conditions over history variables (e.g. the constraint associated to the $work$ port of type $Slave$ in figure 9).

2.3.3 Dynamic Reconfigurable BIP

Dynamic Reconfigurable BIP(DR-BIP) [7] is an extension of BIP conceived to support the development of component-based systems supporting various degree of dynamism, including the ability to change many aspects of the system configuration at run-time.

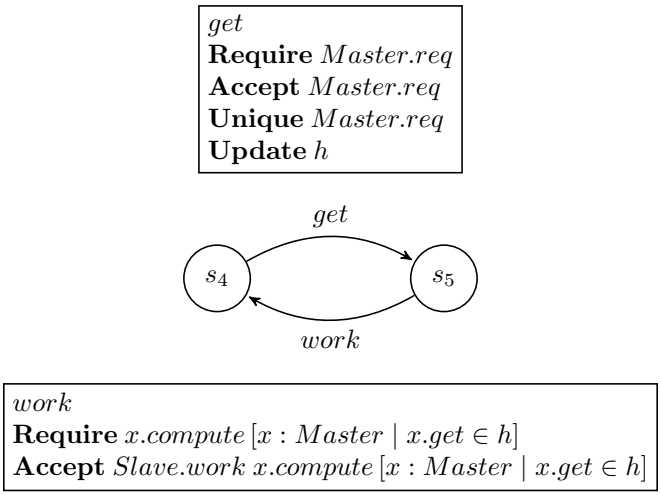


Figure 9: *Slave* type

It has been developed concurrently and independently with the frameworks presented in this Thesis, yet they share many core concepts as they draw inspiration from the same premises and aim at addressing the same problems.

DR-BIP draws inspiration from Dy-BIP in the way it models interactions among BIP components, but uses a distinct approach to handle dynamic changes in the system configuration. More specifically, DR-BIP uses an exogenous coordination paradigm much closer to the ones characterizing ADLs offering strict separation between behavior and architecture.

One of the core-concepts of the framework is the *architectural motif*, which is the minimal building block describing an architecture in DR-BIP. A *motif* is characterized by:

– a set of components, contributing to the *behavior* of the motif;

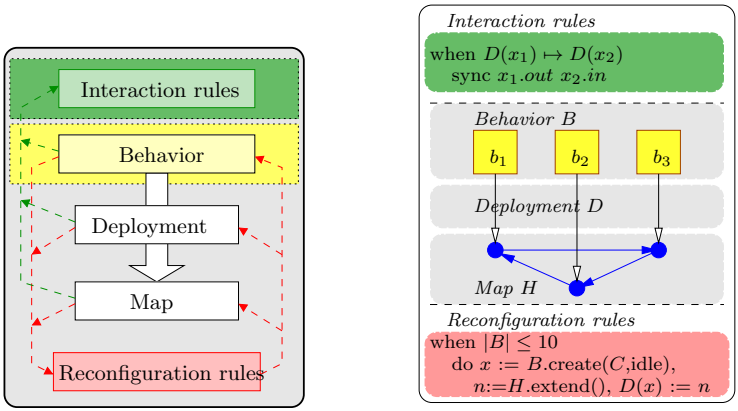
– *interaction rules* for components in the motif;

– *reconfiguration rules* that modify the motif's configuration;

– a graph-like data structure called *map* consisting of interconnected *positions*;

– a *deployment* function that relates components in the motif to positions in its map.

Figure 10 presents the abstract structure of a motif and a simple example.



(a) Abstract structure of a motif **(b)** Example of an architectural motif

Figure 10: Architectural motifs in DR-BIP [7]

Systems are built combining multiple motifs and even "overlapping" them: indeed, the same components can be bound to several motifs at the same time, but they can also *migrate* from one motif to another at run-time.

Each motif encapsulates its own coordination and reconfiguration capabilities in the form of interaction and reconfiguration rules, respectively. These two types of rules that define how a motif evolves determine a dichotomy that is reflected in the operational semantics of DR-BIP systems: the execution of "interaction steps" has no effect on the architecture, which in turn is modified explicitly via "reconfiguration steps" that have no impact on the state of individual components. This concept is summarized in figure 11.

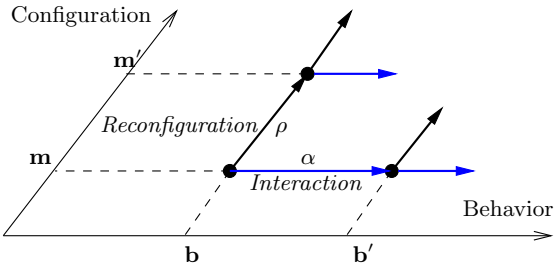


Figure 11: System evolution: reconfiguration vs interaction [7]

System parametrisation is realized not only via maps, but also by *typing* components and motifs. Indeed, DR-BIP is equipped with the notions of *component types* and *motif types*, which serve as blueprints to create new *instances* from a given type.

In addition to "local" reconfiguration actions, DR-BIP provides rules for inter-motif reconfiguration that allow creation/deletion of motif instances and migration of component instances between motifs. These enable dynamic modification of entire architectural styles at run-time, such as migrating all the component instances of a system to another motif instance characterized by completely different interaction/reconfiguration rules or map.

Chapter 3

The L-DReAM framework

This chapter introduces the L-DReAM framework [37], a "light" variant of the DReAM framework [18, 43] for modeling Dynamic Reconfigurable Architectures, which will be presented later in Chapter 4. Both frameworks rely on a logic-based modeling language which is expressive and powerful enough to support different approaches to coordination and provides all the key features required to capture dynamism. L-DReAM differentiates from DReAM mainly by "blurring" the separation between behavior and coordination of components, and by offering a reduced set of parametrization data structures by default.

In L-DReAM a system is a hierarchical structure of components. The latter are characterized by an *interface* of *ports* serving as endpoints for interactions, a *store* of *local variables*, and a *rule* describing components behavior. Furthermore, each non-atomic component hosts a "pool" of components and defines the coordination rules that regulate how members of the pool interact and evolve. The overall structure of systems can also change as components are created and deleted, or leave and join different pools. Components will thus be subject to different coordination rules as they change their position in the hierarchy.

L-DReAM rules include an interaction constraint, modeled as a formula of the Propositional Interaction Logic (PIL) [11], and some operations allowing data transfer as well as more complex reconfigurations of

the component's state. Parametric coordination between classes of components is achieved through the introduction of the concepts of *component types* - blueprints for actual components - and *component instances* created from specific types. Their coordination is characterized by rules in a first order extension to PIL with quantification over instance variables of a given type.

In L-DReAM there is no separation between behavior and coordination of components. In fact, L-DReAM rules are used to coordinate subcomponents in a compound's pool and to characterize both the behavior and the capabilities of components. Furthermore, the possibility of having compounds hosting other compounds in their pool allows to treat every element in the hierarchy uniformly, to the point where the overall system itself is a component.

All this allows L-DReAM to be expressive while boasting a streamlined and uniform operational semantics, making it suitable for theoretical analysis and comparison with other formalisms.

3.1 PIL-based systems

This section reviews some of the fundamental building blocks needed to understand the foundations of L-DReAM focusing on the semantics of Propositional Interaction Logic (PIL) applied to simplified systems where no operation is carried out (i.e., no data transfer and no reconfiguration).

3.1.1 Propositional Interaction Logic (PIL)

At its core, PIL is a "specialized" version of propositional logic where the atoms are *port names* that characterize the interface of components in a system. This allows us to define how components coordinate by shifting the perspective from connectors to *interactions*. In other words, if we define an interaction as a set of ports obtained by composing interfaces of components, PIL can be used to express which interactions are allowed through a logical formula.

Let $\mathcal{P}$ be the domain of ports. The formulas of Propositional Interaction Logic $\Psi \in \mathsf{PIL}(\mathcal{P})$ are defined by the following syntax:

$$\text{(PIL formula)} \quad \Psi ::= \mathbf{true} \mid p \in \mathcal{P} \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \tag{3.1}$$

We will also use the derived logical connectives $\vee$ and $\Rightarrow$ with the classical meaning.

The models of the logic are interactions on the set of ports $\mathcal{P}$. The semantics is defined by the following satisfaction relation $\models$ between an interaction a and a PIL formula:

$$\begin{array}{ll} a \models \mathbf{true} & \text{for any } a \\ a \models p & \text{if } p \in a \\ a \models \Psi_1 \wedge \Psi_2 & \text{if } a \models \Psi_1 \wedge a \models \Psi_2 \\ a \models \neg\Psi & \text{if } a \not\models \Psi \end{array} \tag{3.2}$$

A monomial $\bigwedge_{p\in I} p \wedge \bigwedge_{p\in J} \neg p$ with $I \cap J = \emptyset$ characterizes a set of interactions a such that:

1. the positive terms correspond to *required ports* for the interaction to occur;

2. the negative terms correspond to *inhibited ports* to which the interaction is "closed";

3. the non-occurring terms are *optional ports*.

When the set of optional ports is empty, the monomial is a single interaction and it is characterized by $\bigwedge_{p\in a} p \wedge \bigwedge_{p\notin a} \neg p$.

Given the set of ports $\mathcal{P}$, two formulas $\Psi_1, \Psi_2 \in \mathsf{PIL}(\mathcal{P})$ are *equivalent* (i.e., $\Psi_1 \equiv \Psi_2$) if $a \models \Psi_1 \Leftrightarrow a \models \Psi_2$ for any $a \subseteq 2^{\mathcal{P}}$.

3.1.2 Interacting Components

A system model in PIL is the composition of *interacting components*. Each component B is a transition system characterized by an *interface* P, i.e., a vocabulary of port names, a set of states S, and a set of transitions

between such states labeled with an atom of $\mathsf{PIL}(P)$. Interacting components are completely coordination-agnostic, as there is no additional characterization to ports and states beyond their names (e.g. we do not distinguish between input/output ports or synchronous/asynchronous components).

Definition 3.1.1 (Interacting component)**.** Let $\mathcal{P}$ and $\mathcal{S}$ respectively be the domain of ports and states.

A component is a transition system $B = (P, S, T, s_0)$ where:

- $P \subseteq \mathcal{P}$: finite set of ports;
- $S \subseteq \mathcal{S}$: finite set of states;
- $T \subseteq S \times P \times S$: finite set of transitions $\langle s, p, s' \rangle$, where $p \in P$ and $s, s' \in S$;
- $s_0 \in S$: the initial state.

It is assumed that the sets of ports and states of different components are disjoint.

As a result of participating in an interaction a with port p, a component $B = (P, S, T, s_0)$ can change state from s to s' if there exists a transition $\langle s, p, s' \rangle \in T$. Alternatively if $a \cap P = \emptyset$, then we say that the component stays "idle" not changing its current state while the interaction is performed.

This can be expressed through the following inference rule:

$$\frac{(p \in a \wedge \langle s, p, s' \rangle \in T) \ \vee \ (a \cap P = \emptyset \wedge s' = s)}{s \xrightarrow{a} s'} \tag{3.3}$$

3.1.3 Systems of components

In this simple framework a system specification is characterized by a set of interacting components $B_i = (P_i, S_i, T_i, s_{0i})$ for $i \in [1, n]$. The *state* γ of a system is the set of the current states of each constituent component:

$$\gamma = \{s_i \in S_i\}_{i \in [1..n]} \tag{3.4}$$

Given the finite set of ports of a system $P = \bigcup_{i \in [1..n]} P_i$, an interaction a is any finite subset of P such that:

– every port $p_i \in a$ belongs to a component B_i of the system;

– every component B_i participates in the interaction a at most with one port, i.e., if $p_i \in a$ and $p_j \in a$ (with $p_i \neq p_j$), then $B_i \neq B_j$ for $i, j \in [1..n]$.

The set of all interactions $I(P)$ is a subset of 2^P.

Given a system consisting of components $B_1 \dots B_n$, the set of allowed interactions for the system can be characterized by a formula $\Psi \in \mathsf{PIL}(P)$. The operational semantics of the system from a configuration γ to γ' is then defined by the following rule:

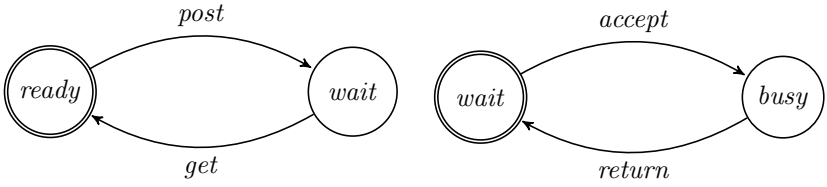
$$\frac{a \models \Psi \qquad s_i \xrightarrow{a} s_i' \quad \text{for } s_i \in \gamma \qquad \gamma' = \{s_i'\}_{i \in [1..n]}}{\gamma \xrightarrow{a} \gamma'} \tag{3.5}$$

where s_i is the current state of component B_i.

Two PIL systems made of the same set of interacting components and characterized by PIL formulas Ψ_1, Ψ_2 are *equivalent* if $\Psi_1 \equiv \Psi_2$.

Example 3.1.1 (Basic client-server interactions)**.** Let us consider a simple system consisting of two components: a *Client* and a *Server*. The interaction between the two components represents the typical "client-server" pattern where the client sends a request to the server, and then the server replies with the requested content. We assume that the exchanges between the two components are fully synchronous.

Figure 12 provides a graphical representation of the LTS of such components.



(a) The *Client* component **(b)** The *Server* component

Figure 12: *Client* and *Server* components

The set of interactions that we allow to model the described behavior at any give state of the system is the following:

$$A = \{\{post, accept\}, \{get, return\}, \emptyset\}$$

where the empty interaction represents the ability of the components to stay idle without interacting.

Instead of writing A explicitly, we can represent it using PIL with the following formula:

$$\begin{aligned}\Psi_1 = &(post \wedge accept) \vee (get \wedge return) \\ &\vee (\neg post \wedge \neg get \wedge \neg accept \wedge \neg return)\end{aligned}$$

An alternative way of expressing the same constraint but using a different "style" is the following:

$$\begin{aligned}\Psi_2 = &(post \Rightarrow accept) \wedge (accept \Rightarrow post) \\ &\wedge (get \Rightarrow return) \wedge (return \Rightarrow get)\end{aligned}$$

It is straightforward to see that $\Psi_2 \equiv \Psi_1$ given the basic properties of propositional logic operators. Nonetheless, we will see that as the complexity of the systems we need to specify grows, choosing to model interactions with one approach or the other can have meaningful implications in the design process.

3.1.4 Disjunctive and Conjunctive styles in PIL

The problem of designing a system that has to evolve through a set of given interactions using PIL as coordination language can be approached in many ways, as there are multiple ways of writing formulas that produce equivalent systems. We distinguish two very different styles that can be adopted when describing allowed interactions, which we call *disjunctive* and *conjunctive*.

The *disjunctive style* approach is based on the idea of starting from an initial formula which does not allow any interaction for the components. From this "false" atom, the capabilities of the components are incrementally *extended* by using the logical "or" $\vee$ to combine formulas that model individual interaction patterns. In other words, each sub-formula can describe an interaction in its entirety. The PIL formula Ψ_1 in example

3.1.1 adopts this style to define which interactions are allowed within the system.

The *conjunctive style* approach starts from the opposite premise, i.e., by assuming that any interaction is allowed. From this "true" atom, the capabilities of the components are incrementally *restricted* by using the logical "and" $\wedge$ to combine formulas that model individual contributions to interaction patterns and the relevant requirements that must hold. To guarantee incremental composability of conjunctive constraints, each formula must always be satisfied by the "idling" interaction $a_0 = \emptyset$, so that there always exists at least one model for the conjunction of any two conjunctive style PIL formulas $\Psi_1 \wedge \Psi_2$.

A methodology for writing conjunctive specifications proposed in [11] considers that each term of the conjunction is a formula of the form $p \Rightarrow \Psi_p$, where the implication is interpreted as a causality relation: for p to be true, it is necessary that the formula Ψ_p holds and this defines the interaction patterns of other components in which the port p needs to be involved. We have already seen an application of this approach when we defined Ψ_2 in example 3.1.1. Other examples are the rendezvous between p_1, p_2 and p_3, modeled by the formula $f_1 = (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3) \wedge (p_3 \Rightarrow p_1)$, and the broadcast from a sending port t towards receiving ports r_1, r_2, which is defined by the formula $f_2 = (t \Rightarrow \mathbf{true}) \wedge (r_1 \Rightarrow t) \wedge (r_2 \Rightarrow t)$. The non-empty solutions of the latter are the interactions $\{t\}$, $\{t, r_1\}$, $\{t, r_2\}$ and $\{t, r_1, r_2\}$.

Notice that, by applying this methodology, we can associate to a component with interface P a constraint $\bigwedge_{p \in P} (p \Rightarrow \Psi_p)$ that characterizes the set of interactions where some port of the component may be involved. Thus, if a system consists of components $B_1, \ldots, B_n$ with interfaces $P_1, \ldots, P_n$ respectively, then the PIL formula $\bigwedge_{i \in [1,n]} \bigwedge_{p \in P_i} (p \Rightarrow \Psi_p)$ expresses a global interaction constraint. Such a constraint can be put in a disjunctive form by simply expanding the implications and rearranging the terms using the associativity and distributivity properties of logical $\wedge, \vee$ operators until the disjunctive normal form is obtained (where monomials characterize global interactions). This means, for instance, that we can have an equivalent disjunctive representation of f_1

as $f'_1 = (p_1 \wedge p_2 \wedge p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_3)$. Notice that the disjunctive form obtained in this manner contains the monomial $\bigwedge_{p \in P} \neg p$, where $P = \bigcup_{i \in [1..n]} P_i$, which is satisfied by the idling interaction. This trivial remark shows that in the PIL framework it is possible to express interaction constraints of each component separately and compose them conjunctively to get global disjunctive constraints.

The inverse operation, i.e., projecting a disjunctive coordination model into the individual contribution for each port of each component, is only possible if the former allows global idling. This condition determines the limits of the conjunctive/compositional approach.

Formally, *a global disjunctive constraint Ψ can be transformed into the conjunction of a series of conjunctive constraints* of the form $p \Rightarrow \Psi_p$, each expressing a contribution to the global interaction from the perspective of each port p in the system, *if and only if* $\emptyset \models \Psi$ (refer to A.1 for the full proof).

To translate a disjunctive formula Ψ satisfying the aforementioned condition into a conjunctive form $\bigwedge_{p \in P} (p \Rightarrow \Psi_p)$, we just need to choose $\Psi_p = \Psi\,[p = \mathbf{true}]$ obtained from Ψ by substituting $\mathbf{true}$ to p. Consider broadcasting from port p to ports q and r (Fig. 13). The possible interactions are characterized by the active ports $\{p\}, \{p, q\}, \{p, r\}, \{p, q, r\}$ and $\emptyset$ (i.e., idling). The disjunctive style formula is: $(\neg p \wedge \neg q \wedge \neg r) \vee (p \wedge \neg q \wedge \neg r) \vee (p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge r) \vee (p \wedge q \wedge r) = (\neg q \wedge \neg r) \vee p$. The equivalent conjunctive formula is: $(q \Rightarrow p) \wedge (r \Rightarrow p)$ that simply expresses the causal dependency of ports q and r from p.

3.2 L-DReAM Syntax and Semantics

L-DReAM can be thought as an enriched version of the simple PIL framework of interacting components presented in section 3.1. Components lose their characterization as labeled transition systems, but are equipped with a set of local variables to store and transfer data, a set of "hosted" components, and a coordination rule that defines both how the component evolves and under which constraints the other components hosted in it can interact. For non-parametric systems, coordination rules are written

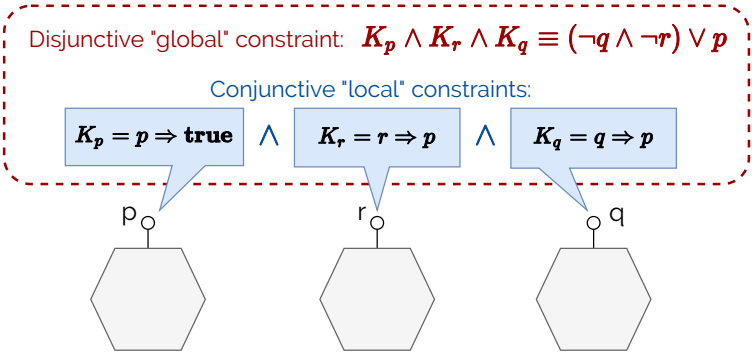


Figure 13: Broadcast example: disjunctive vs conjunctive specification

using a syntax which extends PIL in order to encompass state predicates and operations realizing transfer of data between components. Parametric and fully dynamic systems are then modeled by further expanding the coordination language with first-order quantifiers over component variables and reconfiguration operations that allow to add/remove components from a system or even "migrate" them from one host to another.

This section presents the framework incrementally by first introducing the syntax and the semantics of L-DReAM in the simple scenario of static, non-parametric systems, and then expanding it to encompass parametric system instantiation and dynamic reconfiguration.

3.2.1 Static systems with PILOps

PILOps components and coordination rules

The building blocks of a L-DReAM system are *components*, which are characterized by an *interface* (i.e. a set of port names), a *store* (i.e. a set of local variables), a *pool* (i.e. a set of constituent sub-components) and a behavior *rule*. We refer to components with an empty pool as *atomic*, whereas we call *compound* any non-atomic component (see figure 14).

Definition 3.2.1 (L-DReAM Component)**.** Let $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{X}$ respectively be the domain of components, ports and local variable names. A component is a tuple $c = (P, X, r, \gamma_0)$ with

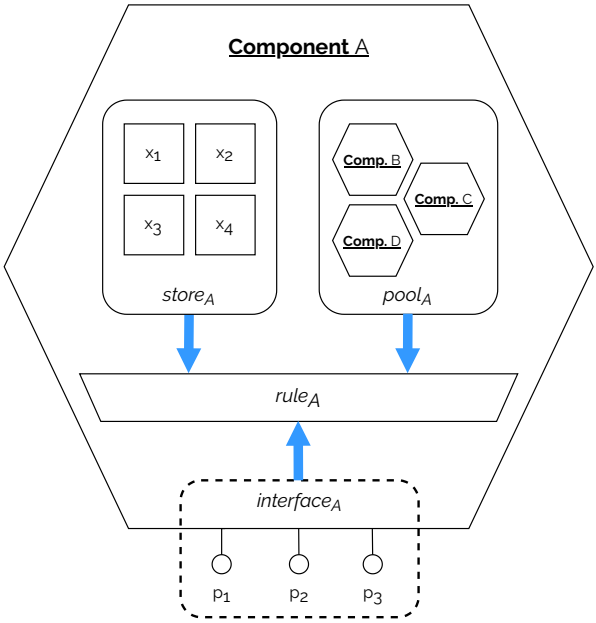


Figure 14: A schematic representation of a L-DReAM compound (i.e., non-atomic component)

- *interface* $P \subseteq \mathcal{P}$: finite set of ports;
- *store* $X \subseteq \mathcal{X}$: finite set of local variables;
- *rule* r: constraint built according to the syntax in (3.6) characterizing the behavior of the component and the coordination between constituents of its pool;
- *initial state* $\gamma_0 \in \Gamma$.

The *state* $\gamma \in \Gamma$ of component c is the tuple $\gamma = (\sigma, C, \Gamma_C)$, where:

- $\sigma : \mathcal{X} \mapsto \mathsf{V}$ is the *valuation function* for the set of local variables X;
- $C \subseteq \mathcal{C}$ is the *pool* of c, i.e., a set of components that c coordinates;
- $\Gamma_C = \{\gamma_{c_i}\}_{c_i \in C}$ and γ_{c_i} is the state of component c_i in the pool of c.

We will use Γ to denote the set of all states. It is assumed that the sets of ports and local variables of different components are disjoint.

Furthermore, we do not allow any component to belong to the pools of two different compounds nor to its own pool.

We will adopt a "dot" notation to express the notion of "ownership" between elements, e.g. $c.p$ will refer to a port p in the interface of c, $c.X$ to the set of local variables of c, and $\gamma_c.C$ to the pool of c for state γ_c.

Given two components c and c', we say that:

- c is a *child* of c' if $c \in \gamma_{c'}.C$;
- c is the *parent* of c' if c' is a *child* of c;
- c is a *descendant* of c' (i.e., $c \in Descendants(c')$) if c is a *child* of c' or if the *parent* of c is a *descendant* of c';
- c is an *ancestor* of c' (i.e., $c \in Ancestors(c')$) if c' is a *descendant* of c.

Behavior and coordination of components are realized only through L-DReAM *rules*, which are expressed in PILOps [18, 43]. PILOps formulas are constructed by combining terms with either the *conjunction operator* $\&$ or the *disjunction operator* $\|$. Each term is essentially a *guarded command* composed of a PIL formula, which encodes a constraint on ports involved in interactions and on states of the associated components, and an operation to be performed when the formula is satisfied. The syntax for PILOps rules is thus defined as follows:

$$\begin{aligned} &\text{(PILOps rule)} && r ::= \Psi \rightarrow \Delta \mid r_1 \,\&\, r_2 \mid\ r_1 \parallel r_2 \\ &\text{(PIL formula)} && \Psi ::= \mathbf{true} \mid p \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \\ &\text{(operation set)} && \Delta ::= \mathbf{skip} \mid\ \{\delta\} \mid \Delta_1 \cup \Delta_2 \end{aligned} \tag{3.6}$$

where:

- operators $\&$ and $\|$ are *associative* and *commutative*, and $\&$ has higher precedence than $\|$;
- $\pi : \Gamma \mapsto \{\mathbf{true}, \mathbf{false}\}$ is a state predicate;
- $\mathbf{skip}$ is the "inaction" (i.e., the identity transformation for a state);

– $\delta : \Gamma \mapsto \Gamma$ is an operation that transforms the state γ of a component;

– the *set union* operator $\cup$ has the traditional meaning, except for how it handles the special operation set **skip**:

$$\mathbf{skip} \cup \Delta = \begin{cases} \mathbf{skip} & \text{if } \Delta \equiv \emptyset \\ \Delta & \text{otherwise} \end{cases} \tag{3.7}$$

As for PIL, the models of the logic are *interactions* a, i.e. finite subsets of the universe of port names $\mathcal{P}$ such that no two ports belong to the same component. To determine whether an interaction is *admissible* for a given PILOps rule, we define a *satisfaction relation* parametric in the state γ of the component "owning" the rule. This relation ignores operations and essentially translates the PILOps rule to a PIL formula by substituting terms $\Psi \to \Delta$ with Ψ and changing operators $\&, \|$ with the logical $\wedge, \vee$. The resulting formula will be a conjunction/disjunction of port names and state predicates that can be checked against a and γ.

Formally, the satisfaction relation is defined by extending (3.2) with the new terms of the syntax (3.6), obtaining the following set of rules:

$$\begin{array}{ll} a \models_\gamma \Psi \to \Delta & \text{if } a \models_\gamma \Psi \\ a \models_\gamma r_1 \,\&\, r_2 & \text{if } a \models_\gamma r_1 \text{ and } a \models_\gamma r_2 \\ a \models_\gamma r_1 \parallel r_2 & \text{if } a \models_\gamma r_1 \text{ or } a \models_\gamma r_2 \\ a \models_\gamma \mathbf{true} & \text{for any } a \\ a \models_\gamma p & \text{if } p \in a \\ a \models_\gamma \pi & \text{if } \pi(\gamma) = \mathbf{true} \\ a \models_\gamma \Psi_1 \wedge \Psi_2 & \text{if } a \models_\gamma \Psi_1 \text{ and } a \models_\gamma \Psi_2 \\ a \models_\gamma \neg\Psi & \text{if } a \not\models_\gamma \Psi \end{array} \tag{3.8}$$

Before discussing how operations are tied to PIL formulas, let us define the *evaluation* of their operands on a given component state. Let $\gamma = (\sigma, C, \Gamma_C)$ be the state of a component c, X its store, Y_j and σ_j respectively the pool and valuation function of component $c_j \in D_c = Descendants(c)$.

The evaluation $\Delta\downarrow_\gamma$ is then defined by the following rules:

$$\begin{aligned} \mathbf{skip}\downarrow_\gamma &\equiv \mathbf{skip} \\ \{\delta\}\downarrow_\gamma &\equiv \left\{\delta\left[\sigma(x)/x\right]_{x\in X}\left[\sigma_j(y)/y\right]_{c_j\in D_c, y\in Y_j}\right\} \\ (\Delta_1\cup\Delta_2)\downarrow_\gamma &\equiv (\Delta_1\downarrow_\gamma)\cup(\Delta_2\downarrow_\gamma) \end{aligned} \tag{3.9}$$

By exhaustively applying (3.9) to an operation set Δ, references to local variables of components are replaced with their evaluation.

Given an admissible interaction a for a component c in state γ, the operations to be performed under a, γ are either:

- for rules combined with "&": the set of all the operations - evaluated under state γ - associated to all PIL formulas if everyone of them holds for (a, γ), or none at all if at least one formula does not;
- for rules combined with "‖": the set of all the operations - evaluated under state γ - associated to PIL formulas satisfied by (a, γ).

The set of operations $[\![r]\!]_{a,\gamma}$ to be performed for r under (a, γ) is defined according to the following rules:

$$\begin{aligned} [\![\Psi\to\Delta]\!]_{a,\gamma} &= \begin{cases} \{\Delta\downarrow_\gamma\} & \text{if } a\models_\gamma \Psi \\ \emptyset & \text{otherwise} \end{cases} \\ [\![r_1\ \&\ r_2]\!]_{a,\gamma} &= \begin{cases} [\![r_1]\!]_{a,\gamma}\cup[\![r_2]\!]_{a,\gamma} & \text{if } a\models_\gamma r_1 \text{ and } a\models_\gamma r_2 \\ \emptyset & \text{otherwise} \end{cases} \\ [\![r_1\parallel r_2]\!]_{a,\gamma} &= [\![r_1]\!]_{a,\gamma}\cup[\![r_2]\!]_{a,\gamma} \end{aligned} \tag{3.10}$$

Notice that from 3.10 follows that if $[\![r]\!]_{a,\gamma_d} = \emptyset$ for any a, it means that γ_d is a *deadlock* state for a system coordinated by rule r, as there is no interaction a that satisfies r.

Rules 3.10 allow to define a notion of *semantic equivalence* between L-DReAM rules:

$$r_1 = r_2 \text{ iff } [\![r_1]\!]_{a,\gamma} = [\![r_2]\!]_{a,\gamma} \text{ for any } a,\gamma \tag{3.11}$$

i.e., two rules r_1, r_2 are *equivalent* if and only if, for any interaction a and component state γ, $[\![r_1]\!]_{a,\gamma} = [\![r_2]\!]_{a,\gamma}$.

Performing the set of operations $\Delta = [\![r]\!]_{a,\gamma} = \{\delta_1, \ldots, \delta_n\}$ on a component c with state γ produces a set of states $\{\gamma'_j\}_{j\in[1..n!]}$. Each γ'_j is the result of the application of an operation δ'_j to γ, where δ'_j is obtained as the composition of all $\delta_i \in \Delta$ in a given order.

Formally, given $\Delta = \{\delta_1, \ldots, \delta_n\}$, let the symmetric group $\mathfrak{S}_\Delta$ of all the $n!$ permutations on the set of operations Δ be:

$$\mathfrak{S}_\Delta = \left\{\left(\delta_{\alpha_j(1)} \cdots \delta_{\alpha_j(n)}\right)\right\}_{j\in[1..n!]} \tag{3.12}$$

where $\alpha_j(i)$ is the j-th permutation of the index i. Then, the application $\Delta(\gamma)$ produces the set $\{\gamma'_j\}_{j\in[1..n!]}$ as follows:

$$\Delta(\gamma) = \left\{\left(\delta_{\alpha_j(1)} \circ \cdots \circ \delta_{\alpha_j(n)}\right)(\gamma)\right\}_{j\in[1..n!]} \tag{3.13}$$

where:

- $\left(\delta_{\alpha_j(1)} \cdots \delta_{\alpha_j(n)}\right) \in \mathfrak{S}_\Delta$ for $j \in [1..n!]$;
- the operator $\circ$ is the function composition (e.g. $f \circ g(x) = f(g(x))$).

Operations δ in PILOps are assignments on local variables of components involved in an interaction. Their syntax is $x := f(y_1, \ldots, y_k)$, where $x \in \mathcal{X}$ is the local variable subject to the assignment and $f : \mathsf{V}^k \mapsto \mathsf{V}$, is a function on local variables $y_1, \ldots, y_k$ ($y_i \in \mathcal{X}$) on which the assigned value depends.

Assuming that operations are always evaluated under a given component state according to (3.9), we define the semantics of the application of assignments $c.x := f(y_1, \ldots, y_k)$ to the state γ of c as:

$$\begin{aligned}
&(c.x := f(y_1, \ldots, y_k))\downarrow_\gamma(\gamma) = \\
&= (c.x := f(v_1, \ldots, v_k))\left(\sigma, C, \Gamma_C\right) \\
&= \left(\sigma\left[c.x \mapsto f(v_1, \ldots, v_k)\right], C, \Gamma_C\right)
\end{aligned} \tag{3.14}$$

where v_i is the value of local variable y_i in state γ.

Notice that by evaluating every operation on the same state γ, the set of states obtained by the application of an operations set Δ to γ according to (3.13) has more than one (distinct) element only if Δ contains two

or more assignments on the same local variable. Indeed, only in this case applying all the assignments in different orders produces different resulting states.

Lastly, we allow the definition of operation sets in the form of *conditional statements* (i.e., "if ... then ... else ...") and *iterative loops* (i.e., "for ... do ..."). These statements are never executed as-is, as they are expanded to sets of actual operations during the evaluation. The syntax and evaluation semantics enriching rules (3.9) are the following:

$$\left(\mathsf{IF}\ (\pi)\ \mathsf{THEN}\ \{\Delta_t\}\ \mathsf{ELSE}\ \{\Delta_e\}\right)\downarrow_\gamma \equiv \begin{cases} \Delta_t\downarrow_\gamma & \text{if } \pi(\gamma) = \mathbf{true} \\ \Delta_e\downarrow_\gamma & \text{otherwise} \end{cases} \tag{3.15}$$

$$\left(\mathsf{FOR}\ \left(i \in [n..m]\right)\ \mathsf{DO}\ \{\Delta_i\}\right)\downarrow_\gamma \equiv \bigcup_{i'=n}^{m} \left(\Delta_i\,[i'/i]\right)\downarrow_\gamma \tag{3.16}$$

Operational semantics of components

Having introduced the syntax of PILOps and formally characterized the semantics of the operations, we can define the evolution of a L-DReAM component c from state γ with the following operational semantics rule:

$$\frac{\begin{array}{l} a \models_\gamma r \qquad \forall \gamma_i \in \gamma.\Gamma_C : \ \gamma_i \xrightarrow{a} \gamma_i' \\ \gamma'' \in [\![r]\!]_{a,\gamma}\left(\gamma.\sigma, \gamma.C, \{\gamma_i'\}_{c_i \in \gamma.C}\right) \end{array}}{\gamma \xrightarrow{a} \gamma''} \ c = (P, X, r, \gamma_0) \tag{3.17}$$

where γ'' is an element of the set of states obtained according to (3.13) by applying the operations $\Delta = [\![r]\!]_{a,\gamma}$ to the intermediate state $\gamma' = \left(\gamma.\sigma, \gamma.C, \{\gamma_i'\}_{c_i \in \gamma.C}\right)$.

Rule (3.17) can be intuitively understood as follows: component c changes state from γ to γ'' through interaction a provided that every component c_i in its pool changes state to γ_i' through a, and that γ'' is the state of c reached from γ' (i.e. the initial state γ of c with the updated states of components in its pool) by applying the operations to be performed for a, γ when a is a model of the rule r under state γ. Rule (3.17) applies to any component. For atomic components with empty pools, the intermediate state will be equal to the initial state, i.e. $\gamma \equiv \gamma'$. For compounds,

it is worth pointing out that a must be an admissible interaction both for the component c, and for every descendant of c.

Notice that the final state γ'' is selected with no specific criterion, therefore the effects of multiple operations on component configurations are generally *non-deterministic*. This behavior is mitigated by the *snapshot semantics* resulting from the evaluation of the operands (3.9) using the initial state γ when computing the set of operations to perform according to (3.10), which prevents the inversion of access/update operations on distinct local variables from producing different outcomes. This limits non-determinism in L-DReAM to the application of multiple assignments on the same local variable, as we have previously mentioned when defining the application of operation sets to component states.

A system model can be seen as a hierarchical tree structure where the leaves are atomic components and the rest of the nodes are compounds that aggregate child components in their pools. As such, the system itself is the *root component* c_0 of this structure, and the initial state γ_0 of a system coincides with the initial state of c_0. By applying rule (3.17) to γ_0 we can characterize the evolution of the whole system. Being the root of the system's hierarchy, c_0 is "orphan" (i.e., has no parent compound) and it is the ancestor of every component in the system. Note that we did not explicitly define a notion of "scope" on port names and local variables, but we shall assume that the rule of a component will only mention ports/variables belonging to itself or to its descendants.

Axioms for PILOps rules

The following axioms hold for PILOps rules:

$$\& \text{ is associative, commutative and idempotent} \tag{3.18}$$

$$\Psi_1 \to \Delta_1 \,\&\, \Psi_2 \to \Delta_2 = \Psi_1 \wedge \Psi_2 \to \Delta_1 \cup \Delta_2 \tag{3.19}$$

$$r \,\&\, \mathbf{true} \to \mathbf{skip} = r \tag{3.20}$$

$$\| \text{ is associative, commutative and idempotent} \tag{3.21}$$

$$\Psi \to \Delta_1 \parallel \Psi \to \Delta_2 = \Psi \to \Delta_1 \cup \Delta_2 \tag{3.22}$$

$$\Psi_1 \to \Delta \parallel \Psi_2 \to \Delta = \Psi_1 \vee \Psi_2 \to \Delta \tag{3.23}$$

Absorption: $r_1 \parallel r_2 = r_1 \parallel r_2 \parallel r_1 \,\&\, r_2$ (3.24)

Distributivity: $r \,\&\, (r_1 \parallel r_2) = r \,\&\, r_1 \parallel r \,\&\, r_2$ (3.25)

Disjunctive Normal Form (DNF):

$$\begin{aligned}&\Psi_1 \to \Delta_1 \parallel \Psi_2 \to \Delta_2 = \\ &= \Psi_1 \wedge \neg\Psi_2 \to \Delta_1 \parallel \Psi_2 \wedge \neg\Psi_1 \to \Delta_2 \parallel \Psi_1 \wedge \Psi_2 \to \Delta_1 \cup \Delta_2\end{aligned} \tag{3.26}$$

Note that PILOps strictly contains PIL as a formula Ψ can be represented by $\Psi \to \mathbf{skip}$. If we focus on its specific $\&$ and $\parallel$ operators though, there are additional similarities and important differences worth pointing out. The operator $\&$ is the extension of conjunction with neutral element $\mathbf{true} \to \mathbf{skip}$ and $\parallel$ is the extension of the disjunction with a distributivity axiom (3.25) and an absorption axiom (3.24). The latter replaces the usual absorption axioms for disjunction and conjunction. Lastly, note that the DNF is obtained by application of the axioms, and there is no conjunctive normal form.

Example 3.2.1 (Server with two Nodes)**.** In example 3.1.1 we started representing a simple client-server system by modeling the synchronous interactions between these two components using PIL. While from the client's perspective a server can be considered a single entity, we know this is just an abstraction offered by a service interface which "hides" a possibly much more complex back-end. We thus expand the given scenario modeling in more details the latter portion of the system where we have a $Server$ component that, given some input data, accepts requests to process it and returns a result. Furthermore, we assume that the requested computation can be easily parallelized, therefore our $Server$ will use two distinct computational $Node$ components to perform the operations.

Since each $Node$ is effectively an integral part of the $Server$ from an external point of view, we will model them as atomic components belonging to the $Server$'s pool. Their specification will be:

$$\begin{aligned}Node_k = \big(&\{receive_k, return_k\}, \{data_k, result_k\}, r_k, \\ &\big(\{data_k \mapsto \mathbf{null}, result_k \mapsto \mathbf{null}\}, \emptyset, \emptyset\big)\big)\end{aligned}$$

meaning that:

– the interface consists of two ports: $receive_k$ (to signal that $Node_k$ can receive new data from the $Server$) and $return_k$ (to send back the result of the computation to the $Server$);

– the store has two local variables: $data_k$ (storing the data to be processed) and $result_k$ (storing the results of the computation);

– the initial state of $Node_k$ has its local variables mapped to **null** and an empty pool (i.e., they are atomic components).

Let $f(x)$ be the function representing the computation that each $Node$ has to perform on input x. Rules r_k can be defined as follows:

$$\begin{aligned} r_k = & data_k = \textbf{null} \wedge result_k = \textbf{null} \wedge receive_k \rightarrow \textbf{skip} \\ & \| \\ & data_k \neq \textbf{null} \rightarrow \{result_k := f(data_k), data_k := \textbf{null}\} \\ & \| \\ & result_k \neq \textbf{null} \wedge return_k \rightarrow \{result_k := \textbf{null}\} \\ & \| \\ & \neg receive_k \wedge \neg return_k \rightarrow \textbf{skip} \end{aligned}$$

This means that a $Node$ can:

– *receive* new input data when both the variable used to store it and the one where the result is saved are empty (**null**);

– compute the result and store it when its $data$ local variable is initialized (resetting the input data);

– *return* the results when computed (resetting the $result$ variable);

– avoid participating in an interaction (negation of all its ports with no associated operation).

Note that a transfer of values between components is achieved by accessing their respective stores through *assignment* operations. In this particular instance, r_k does not involve any transfer of data, as the specification of $Node_k$ simply assumes that someone can assign values to the $data_k$ variables and read values from $result_k$.

The component that will actually perform such operations is the $Server$:

$$\begin{aligned} Server = \big(& \{accept, reduce, return\}, \{input, output\}, r_s, \\ & (\sigma_0, \{Node_1, Node_2\}, \left\{\gamma_0^{N_1}, \gamma_0^{N_2}\right\})) \end{aligned}$$

meaning that:

- the interface of the $Server$ consists of three ports: $accept$ (to accept new requests), $reduce$ (to get the results back from the nodes), and $return$ (to notify that the response to the request is ready);
- the store of the $Server$ has two local variables: $input$ (storing all the data to be processed) and $output$ (storing the complete results of the computation);
- the initial state is characterized by some valuation function σ_0 and by a pool containing $Node_1$ and $Node_2$ (with their respective initial configurations).

Let $split(x, k)$ be a functions that, given an input value x, splits it into two chunks and returns the k-th, and $merge(x, y)$ one that does the opposite returning a value by merging x and y. Rule r_s can be specified as follows:

$$
\begin{aligned}
r_s = & input = \mathbf{null} \wedge accept \rightarrow output := \mathbf{null} \\
& \| \\
& input \neq \mathbf{null} \wedge output = \mathbf{null} \wedge receive_1 \wedge receive_2 \\
& \qquad \rightarrow \{data_1 := split\,(input, 1)\,, data_2 := split\,(input, 2)\} \\
& \| \\
& reduce \wedge return_1 \wedge return_2 \rightarrow \{output := merge\,(result_1, result_2)\} \\
& \| \\
& output \neq \mathbf{null} \wedge return \rightarrow \{input := \mathbf{null}\} \\
& \| \\
& \neg accept \wedge \neg reduce \wedge \neg return \rightarrow \mathbf{skip}
\end{aligned}
$$

We can describe the behavior of the $Server$ induced by the rule r_s as follows:

- it can accept a request ($accept$) when its variable storing the input is not initialized ($input = \mathbf{null}$), resetting at the same time the $output$ variable;
- when its $input$ variable has a value ($input \neq \mathbf{null}$), it can have both $Node_1$ and $Node_2$ synchronize through $receive_k$, initializing their $data_k$ local variables;

– it can synchronize with both nodes through their $return_k$ ports and its $reduce$ port, assigning to $output$ the value obtained by merging the individual results of each $Node_k$;

– it can signal that the response to the original request is ready ($return$) when the $output$ variable is initialized, resetting the $input$ variable in the process;

– it can skip participating in an interaction (negation of all ports with no operation).

Let us consider a concrete case where the $f(x)$ that a $Node_k$ computes is a simple "character count" function (returning the number of characters in x excluding spaces), and the $merge(x, y)$ is the sum $x + y$. To represent the evolution of a system where its root component is the $Server$, we will start from a configuration where its $input$ variable is already initialized:

$$\sigma_0 = \{input \mapsto \text{“hello DReAM!”}, output \mapsto \mathbf{null}\}$$

From here we evaluate which valid interactions A can transform the current state of the system according to (3.17). Since the premise requires that each $a \in A$ can transform the state of each $Node_k$ first, we check for interactions satisfying r_1, r_2 over the interfaces of $Node_1$, $Node_2$. Given that $data_k = \mathbf{null}$ for $k = [1, 2]$, the results are:

$$A_n = \{\emptyset, \{receive_1\}, \{receive_2\}, \{receive_1, receive_2\}\}$$

Next, we consider the $Server$'s rule r_s: for this rule in the current $Server$ state, the admissible interactions are:

$$A_s = \{\emptyset, \{receive_1, receive_2\}, \{reduce, return_1, return_2\}\}$$

Only one of these non-empty interactions can model the $Server$'s rule r_s and both r_1, r_2, that is $a_0 = \{receive_1, receive_2\}$. There are no other admissible interactions in the current state, so this is the one that is performed.

Given that $[\![r_k]\!]_{a_0, Node_k.\gamma_0} \equiv \mathbf{skip}$, the intermediate state produced by the execution of operations associated to each rule of $Node_k$ will be identical to $Server.\gamma_0$.

Then, we evaluate the operations that will be carried out according to the $Server$'s rule r_s:

$$[\![r_s]\!]_{a_0, Server.\gamma_0'} = \{data_1 := split(input, 1), data_2 := split(input, 2)\}$$

By applying the operations to $Server.\gamma_0'$ we obtain the new state $Server.\gamma_1$:

$$\begin{aligned}
Server.\gamma_1 =& \big(\{input \mapsto \text{“hello DReAM!”}, output \mapsto \mathbf{null}\}, \{Node_1, Node_2\}, \\
& \{Node_1.\gamma_1, Node_2.\gamma_1\} \big) \\
Node_1.\gamma_1 =& \big(\{data_1 \mapsto \text{“hello ”}, result_1 \mapsto \mathbf{null}\}, \emptyset, \emptyset \big) \\
Node_2.\gamma_1 =& \big(\{data_2 \mapsto \text{“DReAM!”}, result_2 \mapsto \mathbf{null}\}, \emptyset, \emptyset \big)
\end{aligned}$$

In the new state, the only admissible interaction is now the empty interaction $a_1 = \emptyset$. At $Node$ level, since $data_k \neq \mathbf{null}$, each one will perform:

$$[\![r_k]\!]_{a_1, Node_k.\gamma_1} = \{result_k := f(data_k), data_k := \mathbf{null}\} \tag{3.27}$$

Given that $f(\text{“}hello\text{”}) = 5$ and $f(\text{“}DReAM!\text{”}) = 6$, the new state $Server.\gamma_2$ will be:

$$\begin{aligned}
Server.\gamma_2 =& \big(\{input \mapsto \text{“hello DReAM!”}, output \mapsto \mathbf{null}\}, \{Node_1, Node_2\}, \\
& \{Node_1.\gamma_2, Node_2.\gamma_2\} \big) \\
Node_1.\gamma_2 =& \big(\{data_1 \mapsto \mathbf{null}, result_1 \mapsto 5\}, \emptyset, \emptyset \big) \\
Node_2.\gamma_2 =& \big(\{data_2 \mapsto \mathbf{null}, result_2 \mapsto 6\}, \emptyset, \emptyset \big)
\end{aligned}$$

From state $Server.\gamma_2$, the admissible interactions for $Node_1$ and $Node_2$ become:

$$A_n = \{\emptyset, \{return_1\}, \{return_2\}, \{return_1, return_2\}\}$$

while for the $Server$ component:

$$A_s = \{\emptyset, \{receive_1, receive_2\}, \{reduce, return_1, return_2\}\}$$

which leaves as the only non-empty admissible interaction the set of ports $a_2 = \{reduce, return_1, return_2\}$. As a result, each $Node_k$ will reset its $result_k$ variable with the assignment:

$$[\![r_k]\!]_{a_2, Node_k.\gamma_2} = \{result_k := \mathbf{null}\}$$

The $Server$ will perform $output := 11$ given that

$$merge\,(result_1, result_2) = 5 + 6 = 11$$

producing the new state $Server.\gamma_3$:

$$\begin{aligned}
Server.\gamma_3 =& \big(\{input \mapsto \text{“hello DReAM!”}, output \mapsto 11\}, \{Node_1, Node_2\}, \\
& \{Node_1.\gamma_3, Node_2.\gamma_3\} \big) \\
Node_1.\gamma_3 =& \big(\{data_1 \mapsto \mathbf{null}, result_1 \mapsto \mathbf{null}\}, \emptyset, \emptyset \big) \\
Node_2.\gamma_3 =& \big(\{data_2 \mapsto \mathbf{null}, result_2 \mapsto \mathbf{null}\}, \emptyset, \emptyset \big)
\end{aligned}$$

Now the *Server* can synchronize with an external “client” component through the *return* port and start over. □

3.2.2 Disjunctive and Conjunctive styles in PILOps

The approach shown in section 3.1.4 to write conjunctive-style rules by using logical implications in PIL formulas does not translate directly to PILOps as the semantics of operations is the same of (parallel) guarded commands: at each state one or more guards can be satisfied by the selected interaction, and for all those that do the associated operation is carried out. Instead, the idea behind the conjunctive characterization of admissible interactions is that, in order to guarantee composability, each and every formula must hold for any admissible interaction.

Suppose we have a component c that can interact through its port p only provided that some constraint Ψ_p holds: a causal PIL formula that we could write for a conjunctive style specification is $\Psi = p \Rightarrow \Psi_p$. Furthermore, suppose we want a local variable x of c to increment its value as a result of c interacting through its port p: $\Delta = \{x := x + 1\}$. If we were to build a L-DReAM term by associating Δ as a “consequence” of Ψ, we would obtain $p \Rightarrow \Psi_p \rightarrow \{x := x + 1\}$. This however does not model the intended behavior: operations in Δ are carried out if the formula Ψ holds, and given that Ψ in conjunctive style is an implication of the form $p \Rightarrow \Psi_p$, this rule would cause Δ to be executed for *any* admissible interaction (i.e., even those where p is not involved at all).

Indeed, a minimal contribution to a rule written in conjunctive style needs to combine two PILOps rules: one defining which port of the component is offered for participation in the interaction, the other under which conditions it will participate and which operations will be per-

formed as a result. These can be extended in order to also describe operations triggered by predicates on the state of a component independently from its participation in an interaction, defining a *conjunctive term*.

Definition 3.2.2 (Conjunctive term)**.** Let $c = (P, X, r, \gamma_0)$ be an L-DReAM component according to definition 3.2.1. A *conjunctive term* for c is defined as:

$$\rho \rhd \Psi_\rho \rightarrow \Delta_\rho \triangleq (\neg\rho \rightarrow \mathbf{skip} \parallel \rho \wedge \Psi_\rho \rightarrow \Delta_\rho) \tag{3.28}$$

where:

- $\rho ::= p \in P \mid \pi(X)$ is the *premise* of the term, and is either a port in the interface of c or a predicate on its local variables;
- Ψ_ρ is the *requirement* of the term, which is either $\mathbf{true}$ if $\rho = \pi$ or a PIL formula otherwise;
- Δ_ρ is the *consequence* of the term.

A conjunctive term is *consistent* if its consequence only modifies the state of c or of any of its descendants. That is, Δ_ρ is either $\mathbf{skip}$ or it only contains assignments of the form $x := f(Y)$, with $x \in X$ or $x \in c'.X$ where $c' \in Descendants(c)$.

Definition 3.2.2 allows for two types of conjunctive terms:

1. $p \rhd \Psi_p \rightarrow \Delta_p$: defining the coordination constraint for the involvement of the port p in an interaction and the associated operations;
2. $\pi \rhd \mathbf{true} \rightarrow \Delta_\pi$: defining the operations that transform the state of a component when it satisfies π.

A PILOps rule can be written in conjunctive style by using consistent conjunctive terms as sub-rules, joined via the $\&$ operator.

No additional notation is required to write rules in disjunctive style: it is sufficient to combine base terms of PILOps rules (3.6) with the $\parallel$ operator.

Rules written using the conjunctive style have a direct translation to an equivalent disjunctive style rule. Consider for instance, the combination of two conjunctive terms on ports p and q:

$$(p \rhd \Psi_p \rightarrow \Delta_p) \;\&\; (q \rhd \Psi_q \rightarrow \Delta_q) =$$

$$= (\neg p \to \mathbf{skip} \parallel p \wedge \Psi_p \to \Delta_p) \,\&\, (\neg q \to \mathbf{skip} \parallel q \wedge \Psi_q \to \Delta_q) =$$
$$= \neg p \wedge \neg q \to \mathbf{skip} \parallel p \wedge \neg q \wedge \Psi_p \to \Delta_p \parallel q \wedge \neg p \wedge \Psi_q \to \Delta_q \parallel$$
$$p \wedge q \wedge \Psi_p \wedge \Psi_q \to \Delta_p \cup \Delta_q$$

The disjunctive form obtained from the application of the distributivity axiom (3.25) is the union of four terms corresponding to the canonical monomials on p and q and leading to the execution of operation Δ_p, Δ_q, both or none. It is easy to see that for a set of ports P the conjunctive form

$$\mathop{\&}_{p \in P} (\neg p \to \mathbf{skip} \parallel p \wedge \Psi_p \to \Delta_p) \tag{3.29}$$

is equivalent, according to (3.11), to the disjunctive form

$$\mathop{\Big\|}_{I \cup J = P} \Big(\bigwedge_{p_i \in I} p_i \wedge \Psi_{p_i} \wedge \bigwedge_{p_j \in J} \neg p_j \to \bigcup_{p_i \in I} \Delta_{p_i} \Big) \tag{3.30}$$

where $\bigwedge_{i \in \emptyset} \Psi = \mathbf{true}$, and $\bigcup_{i \in \emptyset} \Delta_{p_i} = \mathbf{skip}$. Similarly, for the conjunction of terms with premise $\pi \in \Pi$ we have:

$$\mathop{\&}_{\pi \in \Pi} (\neg \pi \to \mathbf{skip} \parallel \pi \to \Delta_\pi) \equiv \mathop{\Big\|}_{I \cup J = \Pi} \Big(\bigwedge_{\pi_i \in I} \pi_i \wedge \bigwedge_{\pi_j \in J} \neg \pi_j \to \bigcup_{\pi_i \in I} \Delta_{\pi_i} \Big) \tag{3.31}$$

The converse does not hold. Given a disjunctive specification r it is not always possible to get an equivalent conjunctive one, even if $\emptyset \models_\gamma r$ for any γ. If we have a rule of the form $\|_{k \in K} \Psi_k \to \Delta_k$ over a set of ports P with $\Delta_k \neq \mathbf{skip}$, it can be put in the canonical form, i.e., the union of canonical terms of the form $\bigwedge_{i \in I} p_i \bigwedge_{j \in J} \neg p_j \to \Delta_{IJ}$. It is easy to see that for this form to be obtained as the combination of consistent conjunctive terms, a sufficient condition is that for each port p_i there exists an operation Δ_{p_i} such that $\Delta_{IJ} = \bigcup_{i \in I} \Delta_{p_i}$ and $p_i \rhd \Psi_{p_i} \to \Delta_{p_i}$ is consistent. This condition also determines the limits of the conjunctive/compositional approach for PILOps. That is, in order to express a disjunctive rule with a conjunctive equivalent, it must be possible to deconstruct the set of operations of the former rule in subsets of operations each modifying only the state of a component having a port as premise in the associated conjunctive term.

As an example, let two components c_1, c_2 be:

$$c_1 = \left(\{p\}, \{x\}, \mathbf{true} \to \mathbf{skip}, \gamma_0^1\right)$$
$$c_2 = \left(\{u, v\}, \emptyset, \mathbf{true} \to \mathbf{skip}, \gamma_0^2\right)$$

Consider a system defined as a compound having c_1, c_2 in its pool and characterized by the following disjunctive coordination rule:

$$r = p \wedge u \to x := 1 \parallel p \wedge v \to x := 2 \parallel \neg p \wedge \neg u \wedge \neg v \to \mathbf{skip}$$

The admissible interactions for r are $\{p, u, v\}, \{p, u\}, \{p, v\}$, and the idling. Let us try to transform r into a combination of conjunctive terms, one for each port in the system:

$$r' = p \rhd \Psi_p \to \Delta_p \;\&\; u \rhd \Psi_u \to \Delta_u \;\&\; v \rhd \Psi_v \to \Delta_v$$

Reasoning on the models of the interaction logic, we can easily conclude that:

- $\Psi_p = u \vee v$: port p requires either u or v;
- $\Psi_u = p$: port u requires p;
- $\Psi_v = p$: port v requires p.

The problem of completing the translation of r into a conjunctive specification now becomes choosing the appropriate operation sets for each conjunctive term such that $[\![r]\!]_a = [\![r']\!]_a$ for any a (3.11):

$$\begin{aligned}
[\![r]\!]_{\{p,u,v\}} = [\![r']\!]_{\{p,u,v\}} &\Longrightarrow \Delta_p \cup \Delta_u \cup \Delta_v = \{x := 1, x := 2\} \\
[\![r]\!]_{\{p,u\}} = [\![r']\!]_{\{p,u\}} &\Longrightarrow \Delta_p \cup \Delta_u = \{x := 1\} \\
[\![r]\!]_{\{p,v\}} = [\![r']\!]_{\{p,v\}} &\Longrightarrow \Delta_p \cup \Delta_v = \{x := 2\} \\
[\![r]\!]_{\emptyset} = [\![r']\!]_{\emptyset} &= \mathbf{skip}
\end{aligned}$$

while for any other interaction on ports p, u and v: $[\![r]\!]_a = [\![r']\!]_a = \emptyset$. The only choice that makes r and r' semantically equivalent is $\Delta_p = \mathbf{skip}$, $\Delta_u = \{x := 1\}$ and $\Delta_v = \{x := 2\}$, however this produces two conjunctive terms that are not consistent since both consequences affect a local variable of c_1 but have a port of c_2 as premise.

Example 3.2.2 (Server with two Nodes - conjunctive style)**.** We will now revisit Example 3.2.1 and define coordination rules adopting the conjunctive style. Rules r_k of $Node_k$ can be redefined as follows:

$$
\begin{aligned}
r'_k = & receive_k \rhd data_k = \mathbf{null} \wedge result_k = \mathbf{null} \rightarrow \mathbf{skip} \\
& \& \\
& data_k \neq \mathbf{null} \rhd \mathbf{true} \rightarrow \{result_k := f(data_k), data_k := \mathbf{null}\} \\
& \& \\
& return_k \rhd result_k \neq \mathbf{null} \rightarrow \{result_k := \mathbf{null}\}
\end{aligned}
$$

The $Server$ rule r_s can be rewritten in the conjunctive form r'_s:

$$
\begin{aligned}
r'_s = & accept \rhd input = \mathbf{null} \rightarrow \{output := \mathbf{null}\} \\
& \& \\
& receive_1 \rhd receive_2 \wedge input \neq \mathbf{null} \wedge output = \mathbf{null} \\
& \qquad \rightarrow \{data_1 := split\,(input, 1)\} \\
& \& \\
& receive_2 \rhd receive_1 \wedge input \neq \mathbf{null} \wedge output = \mathbf{null} \\
& \qquad \rightarrow \{data_2 := split\,(input, 2)\} \\
& \& \\
& reduce \rhd return_1 \wedge return_2 \\
& \qquad \rightarrow \{output := merge\,(result_1, result_2)\} \\
& \& \\
& return_1 \rhd reduce \rightarrow \mathbf{skip} \\
& \& \\
& return_2 \rhd reduce \rightarrow \mathbf{skip} \\
& \& \\
& return \rhd output \neq \mathbf{null} \rightarrow \{input := \mathbf{null}\}
\end{aligned}
$$

Notice that each sub-rule in the disjunctive rule r_s coordinating $Node_1$ and $Node_2$ has been decomposed in the highlighted conjunctive sub-rules in r'_s. For instance, $receive_k \rhd receive_{k'} \wedge input \neq \mathbf{null} \wedge output = \mathbf{null} \rightarrow data_k := split\,(input, k)$ models the constraint from the perspective of $Node_k$, whose $receive_k$ port is being offered for interaction. Accordingly, the assignment $data_k := split\,(input, k)$ modifies only the store of $Node_k$

as the premise of the term involves one of its ports. The conjunctive term having the port *reduce* as premise characterizes instead an internal behavior of the *Server* that needs to merge the individual results of both *Node* components to produce the output, and as such it is isolated in a separate sub-rule. □

3.2.3 Parametric architectures and dynamic systems

Now we expand the language introduced in section 3.2.1 allowing L-DReAM to describe dynamic system architectures with an arbitrary (finite) number of components.

To have a modeling language with sufficient expressive power to describe classes of systems with an arbitrary number of components while supporting dynamic reconfiguration of their structure, L-DReAM is enriched on three fronts:

1. The concepts of *component type* and *component instance* are introduced in order to decouple the architecture specification from the instanced system;

2. A first-order extension, with quantifiers over component instances, of the logic used in (3.6) is considered;

3. Appropriate operations to *create* and *delete* component instances, and to *migrate* them from one pool to another are introduced.

Definition 3.2.3 (L-DReAM Component Type)**.** Let $\mathcal{P}$ and $\mathcal{X}$ respectively be the domain of port and local variable names. A component type is a tuple $b = (P, X, r, \gamma_0)$ where:

- *interface* $P \subseteq \mathcal{P}$ is a finite set of ports;
- *store* $X \subseteq \mathcal{X}$ is a finite set of local variables;
- *rule* r is a constraint built according to the syntax in (3.33) that characterizes the behavior of the component instances of this type and the coordination between constituents of their pools;
- *initial state* $\gamma_0 \in \Gamma$ of the form $\gamma_0 = (\sigma_0, \emptyset, \emptyset)$ (i.e., where the pool of b is empty).

A component type can be considered as the blueprint for actual components of a L-DReAM system, which we refer to as *component instances*.

Definition 3.2.4 (L-DReAM Component Instance)**.** Let Cid be the domain of instance identifiers and $b = (P, X, r, \gamma_0)$ be a component type. A *component instance* c of type b, identified by $i \in \mathsf{Cid}$, is a L-DReAM component as defined in (3.2.1) with set of ports P, local variables X and corresponding references in the rule r and state γ_0 indexed with i:

$$c = b\,[i] = (P\,[i]\,, X\,[i]\,, r\,[P\,[i]\,/P]\,[X\,[i]\,/X]\,, \gamma_0\,[X[i]/X]) \tag{3.32}$$

It is assumed that each instance is characterized by a unique identifier, regardless of its type.

Notice that even though we do not require interfaces and stores of different component types to be disjoint sets, the assumption of uniqueness of instance identifiers ensures that interfaces and stores of actual component instances are always disjoint.

In order to have rules sufficiently expressive to model the interactions between arbitrary component instances without any prior knowledge of their identifiers, we equip the language for specifying L-DReAM rules with first-order logic quantifiers in the form of *component instance variable declarations*. The syntax in (3.6) then becomes:

$$\begin{aligned}
&\text{(L-DReAM rule)} && r ::= \Psi \to \Delta \mid D\{r\} \mid r_1 \,\&\, r_2 \mid r_1 \parallel r_2 \\
&\text{(declaration)} && D ::= \forall c : c^*.b \mid \exists c : c^*.b \quad \text{where } c \in c^*.C \\
&\text{(PIL formula)} && \Psi ::= \mathbf{true} \mid p \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \\
&\text{(operation set)} && \Delta ::= \mathbf{skip} \mid \{\delta\} \mid \Delta_1 \cup \Delta_2
\end{aligned} \tag{3.33}$$

Note that conjunctive terms $\rho \rhd \Psi_\rho \to \Delta$ introduced for PILOps in section 3.2.2 can still be used as terms for L-DReAM rules with the interpretation given by (3.28).

A declaration of the form $\forall c : c^*.b$ can be understood as the definition of a variable name c representing component instances of type b in the pool of the component instance c^*. If the scope of a declaration in a rule of a component instance is its own pool, then we simply omit the reference to it and write $\forall c : b$ instead. Similarly, if we do not want to restrict the

type of the component instance in the declaration, we simply omit it and write $\forall c : c^*$ (or $\forall c$ if the scope of the variable c is the pool of the same component). $\exists c : c^*.b$ can be interpreted similarly.

Note that ports p and predicates π in PIL formulas can now be parametric with respect to the instance variables defined in the enclosed declaration (e.g. a rule with a declaration $\forall c : b$ can refer to a port p of all component instances c that match the declaration, using the dot notation $c.p$).

Under the assumption that L-DReAM systems have a finite number of component instances, each declaration in the rule of a component instance can be removed by transforming the rule itself into a combination of rules (via the $\&$ and $\|$ operators for the universal and existential quantifiers, respectively) where the instance variable is replaced with the actual component instances of the given type in the given pool. We refer to this transformation as the *declaration expansion* $\langle\langle r \rangle\rangle_\gamma$ of rule r of a component instance with state $\gamma = (\sigma, C, \Gamma_C)$, which is formally defined by the following rules:

$$
\begin{aligned}
\langle\langle \Psi \to \Delta \rangle\rangle_\gamma &= \Psi \to \Delta \\
\langle\langle \forall c : c^*.b\{r\} \rangle\rangle_\gamma &= \mathop{\&}_{b[i] \in c^*.C} \langle\langle r\,[b\,[i]\,/c] \rangle\rangle_\gamma \\
\langle\langle \forall c : c^*\{r\} \rangle\rangle_\gamma &= \mathop{\&}_{\forall b} \langle\langle \forall c : c^*.b\{r\} \rangle\rangle_\gamma \\
\langle\langle \exists c : c^*.b\{r\} \rangle\rangle_\gamma &= \mathop{\|}_{b[i] \in c^*.C} \langle\langle r\,[b\,[i]\,/c] \rangle\rangle_\gamma \\
\langle\langle \exists c : c^*\{r\} \rangle\rangle_\gamma &= \mathop{\|}_{\forall b} \langle\langle \exists c : c^*.b\{r\} \rangle\rangle_\gamma \\
\langle\langle r_1 \,\&\, r_2 \rangle\rangle_\gamma &= \langle\langle r_1 \rangle\rangle_\gamma \,\&\, \langle\langle r_2 \rangle\rangle_\gamma \\
\langle\langle r_1 \,\|\, r_2 \rangle\rangle_\gamma &= \langle\langle r_1 \rangle\rangle_\gamma \,\|\, \langle\langle r_2 \rangle\rangle_\gamma
\end{aligned}
\tag{3.34}
$$

where $r\,[b\,[i]\,/c]$ is the rule r after applying the substitution of the instance variable c with the actual instance $b\,[i]$.

Example 3.2.3 (Server with n Nodes - declaration expansion)**.** Let us expand Example 3.2.1 and 3.2.2 by considering a scenario where we have a variable pool of *Node* component instances that one *Server* instance can use to handle the computation.

We will define the component types $Server$ and $Node$ as follows:

$$Server = (\{accept, reduce, return\}, \{input, output\}, r_s, \gamma_0)$$
$$Node = (\{receive, return\}, \{id, data, result\}, r_n, \gamma_0)$$

where we introduced the local variable id to keep track of the identifier of $Node$ instances ready to process their $Server$'s input, and the $output$ variable is now an array of values (where $output\,[i]$ represents the i-th element of the array).

Consider now a system with the same initial configuration described in Example 3.2.1, where we have one $Server$ instance - $Server[0]$ - and two $Node$ instances - $Node[1], Node[2]$. The initial state γ_0 of $Server[0]$ will be:

$$Server[0].\gamma_0 = \big(\{input \mapsto \text{“hello DReAM!”}, output \mapsto \mathbf{null}\}, \\ \{Node[1], Node[2]\}, \{Node[1].\gamma_0, Node[2].\gamma_0\} \big)$$

Let us define a fragment r of rule r_s of the $Server$ component type modeling how $Node$ instances are fed input data to be processed using the conjunctive style:

$$r = \forall c : Node \big\{ c.receive \rhd input \neq \mathbf{null} \wedge output = \mathbf{null} \\ \rightarrow \{c.data := split\,(input, \mathsf{poolSize}(\texttt{this}), c.id)\} \big\}$$

where we extended the *split* function in order to parametrize the splitting degree with the number of components in the $Server$'s pool (referenced through the function $\mathsf{poolSize}(c)$). The declaration expansion of rule r under state $Server[0].\gamma_0$ is performed by generating a new rule for each $Node$ instance in the pool of $Server$ and substituting it in place of the instance variable c:

$$\big\langle\!\big\langle r \big\rangle\!\big\rangle_{\gamma_0} = Node[1].receive \rhd input \neq \mathbf{null} \wedge output = \mathbf{null} \\ \rightarrow \{Node[1].data := split\,(input, \mathsf{poolSize}(Server[0]), Node[1].id)\} \\ \& \\ Node[2].receive \rhd input \neq \mathbf{null} \wedge output = \mathbf{null} \\ \rightarrow \{Node[2].data := split\,(input, \mathsf{poolSize}(Server[0]), Node[2].id)\}$$

□

To encompass the additional step of declaration expansion in the description of the behavior of a L-DReAM component, the inference rule

(3.17) used to describe the operational semantics of the static version of the language needs to be modified accordingly:

$$\frac{\begin{array}{c}\forall c_i \in c.C:\ c_i.\gamma \xrightarrow{a} c_i.\gamma' \qquad c.\gamma' = \left(c.\sigma, c.C, \{c_i.\gamma'\}_{c_i \in c.C}\right) \\ a \models_{c.\gamma} \left\langle\!\left\langle c.r \right\rangle\!\right\rangle_{c.\gamma} \qquad c.\gamma'' \in [\![\left\langle\!\left\langle c.r \right\rangle\!\right\rangle_{c.\gamma}]\!]_{a,c.\gamma}\left(c.\gamma'\right)\end{array}}{c.\gamma \xrightarrow{a} c.\gamma''} \tag{3.35}$$

Rule (3.35) now requires that, at each state of a component instance, the problem of finding an interaction that satisfies its coordination rule is solved after it is reduced to the static, non-parametric case by expanding the declarations.

Semantics of reconfiguration operations

Since the higher-order language now allows us to express constraints without prior knowledge of the individual component instances in a system, reconfiguration operations are extended in order to allow dynamic variations in the population of component instances. We thus have three new operations:

- $\mathsf{create}\,(c\!:\!b, c_s, g)\,[\Delta_c]$: to create a new component instance of type b, add it to the pool of component instance c_s, and bind it to the instance variable c within the scope of the operation set Δ_c which is then executed, where g is a *generator* attribute which can be used to distinguish multiple creation operations with the same parameters;

- $\mathsf{delete}(c)$: to delete the component instance c;

- $\mathsf{migrate}\,(c, c_s)$: to migrate component instance c to the pool of c_s (removing it from the pool where it previously belonged).

Note that the "create" operation can be written using the shorthand notation $\mathsf{create}(b, c_s, g)$ when no chained operation has to be performed after the new instance is created (i.e., $\mathsf{create}(b, c_s, g) \equiv \mathsf{create}\,(c\!:\!b, c_s, g)\,[\mathbf{skip}]$).

To formalize the semantics of the application of the new operations on a component state, let us first define the "binding" support operation:

$$\mathsf{bind}\,(c, c_s)\,(\hat{c}.\gamma) = \begin{cases} (\sigma, C \oplus c, \Gamma_C \oplus c.\gamma) & \text{if } c_s = \hat{c} \\ (\sigma, C, \Gamma'_C) & \text{otherwise} \end{cases} \tag{3.36}$$

where:

- $\Gamma'_C = \{\mathsf{bind}\,(c, c_s)\,(\gamma_i) \mid \gamma_i \in \Gamma_C\}$;
- $S \oplus s \equiv S \cup \{s\}$ is the set addition operation.

Note that this operation, which adds c to the pool of c_s, is not explicitly used within L-DReAM rules as its use can produce states where a component belongs to the pool of two different compounds or to its own pool (which we do not allow).

Having defined the support operation bind, the state transformation induced by the application of each operation on the component state $\hat{c}.\gamma = (\sigma, C, \Gamma_C)$ can now be formalized as follows:

$$\mathsf{create}\,(c\!:\!b, c_s, g)\,\left[\Delta_c\right]\,(\hat{c}.\gamma) = (\Delta_c\,[b[f]\,/c])\!\downarrow_{\gamma'}(\gamma') \tag{3.37}$$

where:

- $f \in \mathsf{Cid}$ is a *fresh* instance identifier;
- $\gamma' = \mathsf{bind}\,(b[f]\,, c_s)\,(\hat{c}.\gamma)$.

$$\mathsf{delete}(c)\,(\hat{c}.\gamma) = \begin{cases} (\sigma, C \ominus c, \Gamma_C \ominus c.\gamma) & \text{if } c \in C \\ (\sigma, C, \Gamma'_C) & \text{otherwise} \end{cases} \tag{3.38}$$

where:

- $\Gamma'_C = \{\mathsf{delete}(c)\,(\gamma_i) \mid \gamma_i \in \Gamma_C\}$;
- $S \ominus s \equiv S \setminus \{s\}$ is the set removal operation.

$$\mathsf{migrate}\,(c, c_s)\,(\hat{c}.\gamma) = \mathsf{bind}\,(c, c_s)\,(\mathsf{delete}(c)\,(\hat{c}.\gamma)) \tag{3.39}$$

Notice that the generator attribute g has no effect on the semantics of the "create" operation (3.37), but contributes to the (implicit) structural equivalence between operations. Formally:

$$\begin{gathered}\mathsf{create}\,(c_1' : b_1, c_1, g_1)\left[\Delta_{c_1'}\right] \equiv \mathsf{create}\,(c_2' : b_2, c_2, g_2)\left[\Delta_{c_2'}\right] \\ \Downarrow \\ b_1 = b_2 \text{ and } c_1 = c_2 \text{ and } g_1 = g_2\end{gathered} \tag{3.40}$$

Different choices for this attribute allow to define whether a L-DReAM rule can create multiple instances of the same component type in the same pool. This can be used to different effect. Consider as an example the following rules:

$$\begin{aligned} r_1 =& \forall c : c_0.b\big\{\mathbf{true} \rightarrow \{\mathsf{create}(b, c_0, 0)\}\,\big\} \\ r_2 =& \forall c : c_0.b\big\{\mathbf{true} \rightarrow \{\mathsf{create}(b, c_0, c.id)\}\,\big\} \end{aligned}$$

Assume that c_0 hosts two component instances of type b in its pool in its current state γ, respectively with instance identifier equal to 1 and 2. The declaration expansion of these two rules in γ according to (3.34) produces:

$$\begin{aligned} \big\langle\!\big\langle r_1 \big\rangle\!\big\rangle_\gamma &= \mathbf{true} \rightarrow \{\mathsf{create}(b, c_0, 0)\}\,\&\,\mathbf{true} \rightarrow \{\mathsf{create}(b, c_0, 0)\} \\ \big\langle\!\big\langle r_2 \big\rangle\!\big\rangle_\gamma &= \mathbf{true} \rightarrow \{\mathsf{create}(b, c_0, 1)\}\,\&\,\mathbf{true} \rightarrow \{\mathsf{create}(b, c_0, 2)\} \end{aligned}$$

If we now compute the respective sets of operations to be performed for the two rules according to (3.10), we get:

$$\begin{aligned} [\![\big\langle\!\big\langle r_1 \big\rangle\!\big\rangle_\gamma]\!]_\gamma &= \{\mathsf{create}(b, c_0, 0)\} \\ [\![\big\langle\!\big\langle r_2 \big\rangle\!\big\rangle_\gamma]\!]_\gamma &= \{\mathsf{create}(b, c_0, 1)\,, \mathsf{create}(b, c_0, 2)\} \end{aligned}$$

Indeed, the two rules could have been interpreted as follows:

- r_1: if c_0 hosts at least an instance of type b in its pool, then create another instance;
- r_2: for every instance of type b hosted in the pool of c_0, create a new instance.

The adoption of a snapshot semantics guarantees that component references are always resolved according to the snapshot, so any operation can still be carried out even if a delete/migrate operation on the involved instance is executed before it. This means, for example, that the execution of the set of operations $\Delta = \{\mathsf{delete}(c)\,, c'.x := c.x\}$ produces a state where c is no longer present and variable $c'.x$ has the same value that $c.x$ had prior to being deleted, regardless of the order in which the two operations are resolved.

Conjunctive style in L-DReAM

The full L-DReAM framework relies on the same conjunctive term (3.28) defined for PILOps in section 3.2.2 to build coordination rules using the conjunctive style.

The notion of being "consistent", which we attributed to conjunctive terms with operations that only modify the state of the component involved in their premise or of any of its descendants, still applies.

Considering the reconfiguration operations introduced in the dynamic framework, a conjunctive term on component $\hat{c}$ of the form $\rho_{\hat{c}} \rhd \Psi_{\rho_{\hat{c}}} \rightarrow \Delta_{\rho_{\hat{c}}}$ associated to the rule of component c_r is consistent if its consequence $\Delta_{\rho_{\hat{c}}} \equiv \mathbf{skip}$ or if it only contains operations δ that verify:

- $\mathsf{create}\,(c\!:\!b, c_s, g)\,\big[\Delta_c\big]$: $c_s = \hat{c}$, and $\rho_{\hat{c}} \rhd \mathbf{true} \rightarrow \Delta_c$ is consistent;
- $\mathsf{delete}(c)$: $c \in \mathit{Descendants}(\hat{c}) \oplus \hat{c}$;
- $\mathsf{migrate}\,(c, c_s)$: $c = \hat{c}$, and $c_s \in \mathit{Descendants}(c_r) \oplus c_r$.

Example 3.2.4 (Server with dynamic instantiation of Nodes)**.** Recall the extended scenario presented by Example 3.2.3. We will now integrate the definition of the *Server* and *Node* component types with their L-DReAM rules using the conjunctive style. Rule r_n of the type *Node* will be essentially the same as r_k in Example 3.2.2:

$$
\begin{aligned}
r_n = & \mathit{receive} \rhd \mathit{data} = \mathbf{null} \wedge \mathit{result} = \mathbf{null} \rightarrow \mathbf{skip} \\
& \& \\
& \mathit{data} \neq \mathbf{null} \rhd \mathbf{true} \rightarrow \{\mathit{result} := f(\mathit{data}), \mathit{data} := \mathbf{null}\} \\
& \&
\end{aligned}
$$

$$return \rhd result \neq \mathbf{null} \rightarrow \{result := \mathbf{null}\}$$

Since all the ports and local variables mentioned in r_n will be local to each *Node* instance, no scoping or quantification is needed.

On the other hand r_s will now have to deal with an arbitrary number of nodes. Recall that the service that we want our *Server* to provide is the "character count" function. Let us define r_s as the conjunction of the following six sub-rules $r_1 \& \ldots \& r_6$:

$$\begin{aligned} r_1 = \forall c : Node \big\{ & accept \rhd input = \mathbf{null} \\ & \rightarrow \{output := \mathbf{null}, \mathsf{delete}(c)\} \big\} \end{aligned}$$

Rule r_1 implements the *output* reset of the *Server* as in the non-parametric case of Example 3.2.2, but it also deletes every *Node* instance.

$$\begin{aligned} r_2 = input \neq \mathbf{null} \wedge \mathsf{poolSize}(\mathtt{this}) = 0 \rhd \mathbf{true} \\ \rightarrow \big\{ \mathsf{FOR}\ \big(i = 1..length(input)/5\big)\ \mathsf{DO}\ \big\{ \\ \mathsf{create}\,(c : Node, \mathtt{self}, i)\ \big[c.id := i\big] \big\} \big\} \end{aligned}$$

Rule r_2 is completely new to the parametric variant, and it is used to create as many *Node* instances as needed (e.g. in this case one every five characters in *input*).

$$\begin{aligned} r_3 = \forall c : Node \big\{ & c.receive \rhd input \neq \mathbf{null} \wedge output = \mathbf{null} \\ & \rightarrow c.data := split\,(input, \mathsf{poolSize}(\mathtt{this})\,, c.id) \big\} \end{aligned}$$

Rule r_3 is exactly the same rule as r discussed in Example 3.2.3 describing how *Node* components receive input data from the *Server*.

$$r_4 = \forall c : Node \big\{ reduce \rhd c.return \rightarrow \{output\,[c.id] := c.result\} \big\}$$

Rule r_4 characterizes how processed data is collected by the *Server* by collecting the results in its *output* local variable.

$$r_5 = return \rhd output \neq \mathbf{null} \rightarrow \{input := \mathbf{null}\}$$

Rule r_5 is left unchanged from the non-parametric case.

$$\begin{aligned} r_6 = {} & \forall c : Node \big\{ \forall c' : Node \big\{ c.receive \rhd c'.receive \rightarrow \mathbf{skip} \big\} \big\} \\ & \& \\ & \forall c : Node \big\{ c.return \rhd reduce \rightarrow \mathbf{skip} \big\} \end{aligned}$$

Lastly, r_6 enforces strong synchronization between all $Node$ instances when interacting through the $receive$ port, and between the $Server$ and all its nodes through the $return$ and $reduce$ ports (the latter is, in fact, the result of the combination of the second conjunctive term of r_6 with r_4). □

3.3 Encoding other formalisms

The "loose" characterization of individual components and the freedom to hierarchically compose them makes L-DReAM extremely flexible and capable of describing a variety of systems and programming paradigms.

Consider for example a labeled transition system (S, L, T) defined over the states S, labels L and transitions $T \subset S \times L \times S$. One of the possible modeling of an LTS as a L-DReAM atomic component $c = (P, X, r, \gamma_0)$ is the following:

- every label $l \in L$ has a corresponding port $p_l \in P$;
- every state $s \in S$ has a corresponding local variable $x_s \in X$;
- for every transition $\langle s_i, l_i, s_i' \rangle \in T$ there is a rule $r = \|_i \left(x_{s_i} \wedge p_{l_i} \rightarrow \left\{ x_{s_i} := \mathbf{false}, x_{s_i'} := \mathbf{true} \right\} \right)$.

L-DReAM can also be used in an imperative, sequential fashion. To do so, one possibility is to simply add a "program counter" local variable pc in the store of each component type and conjunct the appropriate predicate over it in the requirements of each conjunctive term describing its behavior:

$$r_{seq} = \underset{i}{\&} \left(p_i \rhd (\Psi_i' \wedge pc = i) \rightarrow \Delta_i \cup \{ pc := pc + 1 \} \right) \tag{3.41}$$

More generally, by using the conjunctive style L-DReAM can be used as an endogenous coordination language comparable to process calculi relying on a single associative parallel composition operator. Consider for instance the CCS [40] process P_0 defined as the parallel composition of two processes P_1 and P_2 that can synchronize over the actions q and u:

$$P_0 = P_1 \mid P_2$$

$$P_1 = q.\bar{u}.0 \qquad P_2 = \bar{q}.u.0$$

where 0 is the inaction. To translate this simple system in L-DReAM we define three components c_i, each one modeling the process P_i:

$$\begin{aligned} c_0 &= \left(\{\tau\}, \emptyset, r_0, \left(\emptyset, \{c_1, c_2\}, \{\gamma_{c_1}, \gamma_{c_2}\}\right)\right) \\ c_1 &= \left(\{q, \bar{u}\}, \{pc_1\}, r_1, \gamma_{c_1}\right) \\ c_2 &= \left(\{\bar{q}, u\}, \{pc_2\}, r_2, \gamma_{c_2}\right) \end{aligned}$$

where the pool of c_0 hosts c_1 and c_2.

Rules r_1, r_2 will model a simple sequential process according to (3.41):

$$\begin{aligned} r_1 = {} & q \rhd (pc_1 = 1) \to \{pc_1 := 2\} \\ & \& \\ & \bar{u} \rhd (pc_1 = 2) \to \{pc_1 := 3\} \\ r_2 = {} & \bar{q} \rhd (pc_2 = 1) \to \{pc_2 := 2\} \\ & \& \\ & u \rhd (pc_2 = 2) \to \{pc_2 := 3\} \end{aligned}$$

The rule r_0 of the root component c_0 will instead characterize the semantics of the CCS parallel composition, which provides that two parallel processes can synchronize over the matching actions (in this case represented by the interactions $\{q, \bar{q}\}$ and $\{u, \bar{u}\}$) causing the system to perform an internal action τ:

$$r_0 = \tau \rhd (q \wedge \bar{q}) \vee (u \wedge \bar{u}) \to \mathbf{skip}$$

Notice that r_0 still allows interleaving between c_1 and c_2 just like for P_0.

Another way of modeling communicating sequential processes in L-DReAM is to equip every component with two ports in, out and characterize the communication channel (or the action label in CCS) with a specific local variable $chan$. This allows to model even more complex synchronization mechanisms of calculi like π-calculus [41] and one of its main features: *channel mobility*. Let us consider a simple process that we will call again P_0, representing the parallel composition of three processes

P_1, P_2 and P_3:

$$P_0 = P_1 \mid P_2 \mid P_3$$
$$P_1 = q(x).\bar{x}\langle v\rangle.0 \qquad P_2 = \bar{q}\langle u\rangle.0 \qquad P_3 = u(y).0$$

The idea is that P_1 wants to communicate with P_3, but initially can only communicate with P_2. Through the output/input pair of actions $\bar{q}\langle u\rangle$ and $q(x)$, P_2 can send the channel u to P_1 which then allows it to send a message to P_3 by binding u to x.

To represent this system in L-DReAM, we will first define the root component type b_0 as before:

$$b_0 = \left(\{\tau\}, \emptyset, r_0, (\emptyset, \emptyset, \emptyset)\right)$$

To model the other processes, we will instead define three component types b_1, b_2 and b_3:

$$b_1 = \left(\{in, out\}, \{pc, chan, val, x\}, r_1, \gamma_0^1\right)$$
$$b_2 = \left(\{in, out\}, \{pc, chan, val\}, r_2, \gamma_0^2\right)$$
$$b_3 = \left(\{in, out\}, \{pc, chan, y\}, r_3, \gamma_0^1\right)$$

where the initial states γ_0^1, γ_0^2 and γ_0^3 are respectively:

$$\gamma_0^1 = \left(\{pc \mapsto 1, chan \mapsto \text{“}q\text{”}, val \mapsto \textbf{null}, x \mapsto \textbf{null}\}, \emptyset, \emptyset\right)$$
$$\gamma_0^2 = \left(\{pc \mapsto 1, chan \mapsto \text{“}q\text{”}, val \mapsto \text{“}u\text{”}\}, \emptyset, \emptyset\right)$$
$$\gamma_0^3 = \left(\{pc \mapsto 1, chan \mapsto \text{“}u\text{”}, y \mapsto \textbf{null}\}, \emptyset, \emptyset\right)$$

Rules r_1, r_2 and r_3 will again model sequential processes, but will now also handle channel names and value output:

$$r_1 = in \rhd (pc = 1) \to \{x := val,\ chan := x,\ val := \text{“}v\text{”},\ pc := 2\}$$
$$\quad \& \quad out \rhd (pc = 2) \to \{pc := 3\}$$
$$r_2 = out \rhd (pc = 1) \to \{pc := 2\}$$
$$r_3 = in \rhd (pc = 1) \to \{pc := 2\}$$

The assignments on the $chan$ variable here are used to indicate the channel name associated with the preceding interaction constraint, while the val variable is used to store the channels being passed.

Synchronization and actual value passing is instead implemented by rule r_0 of the root component c_0:

$$\begin{aligned} r_0 = {} & \exists c \big\{ \exists c' \big\{ \tau \rhd c.in \wedge c'.out \rightarrow \mathbf{skip} \big\} \big\} \\ & \& \\ & \forall c \big\{ \exists c' \big\{ c.in \rhd c'.out \wedge c.chan = c'.chan \rightarrow \{ c.val := c'.val \} \big\} \big\} \end{aligned}$$

This rule characterizes both the behavior of the root component type b_0 - which performs τ if two instances of any type interact via ports in and out - and the coordination between components in its pool - i.e., for one to interact with port in there must be another one participating with port out with matching values for their local variables $chan$. For the sake of simplicity, we are omitting further parts of rule r_0 to enforce only binary synchronization between component instances.

L-DReAM can be also used to model capabilities of many other process algebras relying on more complex and flexible strategies to realize communication and coordination between components than simple channels matching. For instance, specifications written with the AbC calculus [1, 2] can be translated to L-DReAM preserving most of the features of the source language.

In AbC, a system is a set of parallel components equipped with a set of *attributes*. Communication happens in a broadcast fashion with the caveat that only components that satisfy some predicates over specific attributes do receive the message given that also the sender satisfies other predicates. The core actions of the language that characterize its communication paradigm are the *input* and *output* actions:

$$\begin{aligned} (input) \quad & \Pi_1 \, (\tilde{x}) \\ (output) \quad & (\tilde{E}) @ \Pi_2 \end{aligned}$$

where $\tilde{x}$ is a sequence of "placeholders" for the received values and $\tilde{E}$ is a sequence of expressions representing the values being sent. Π are

predicates, which can be defined over attributes only (i.e. for output actions) or also over received values (i.e. for input actions). A simple example of matching input/output actions in AbC could be:

- $(x > 1 \wedge color = blue)(x, y)$: bind two values to variables x, y from messages having $x > 1$ and coming from components with attribute $color$ equal to $blue$;
- $(2, 0)@(color = red)$: send values $2, 0$ to all components with attribute $color$ equal to red.

Broadcast communication can be easily implemented in L-DReAM. To maintain uniformity with AbC where actions are defined at component level, we can adopt the conjunctive style and define two rules that implement its input/output actions:

$$\begin{array}{rl} (input) & \forall c\{\exists c'\{c.in \rhd c'.out \wedge \Pi_1(c'.\tilde{E}) \rightarrow \{c.\tilde{x} := c'.\tilde{E}\}\}\} \\ (output) & \forall c\{\exists c'\{c.out \rhd c'.in \wedge \Pi_2 \rightarrow \mathbf{skip}\}\} \end{array}$$

where we assume that:

- the interface of all component types includes two ports in, out;
- the store of all component types contains local variables describing attributes, values being passed through messages, and the variables that are bound to values being passed;
- the predicates Π_1, Π_2 are equivalent to the respective AbC counterparts, with the appropriate references to local variables modeling attributes and values being passed.

Going back to the simple example mentioned above, we could translate the given pair of input/output actions as:

$$\begin{array}{c} (x > 1 \wedge color = blue)(x, y) \\ \downarrow \\ \forall c\{\exists c'\{c.in \rhd (c'.out \wedge c'.x > 1 \wedge c'.color = blue) \\ \rightarrow \{c.x := c'.x, c.y := c'.y\}\}\} \end{array}$$

$$(2,0)@(color = red)$$
$$\downarrow$$
$$\forall c\{\exists c'\{c.out \rhd (c'.in \wedge c'.color = red) \rightarrow \mathbf{skip}\}\}$$

There are many more nuances to AbC that we are not representing here, such as the fact that the in action is blocking while the out action is not. To model this, we need to combine the previously mentioned encoding of the in and out actions with the approach described earlier to simulate sequential processes in L-DReAM.

Chapter 4

The DReAM framework

The L-DReAM framework introduced in Chapter 3 provides a flexible approach to the specification of dynamic and reconfigurable systems. The light structure imposed on components and compounds makes it very close to a minimal extension to the language used to define its rules, which allows to study it effectively with little overhead. However, the design of most "real" systems does not necessarily benefit from the simplicity of the adopted formalism. Specialized elements of the language addressing common use cases and specific needs can indeed ease the task of system modeling.

This Chapter presents DReAM (Dynamic Reconfigurable Architecture modeling) [18, 43], a specialized version of L-DReAM that is equipped with features supporting effective design of dynamic system architectures. DReAM uses the same logic-based modeling language of its "light" counterpart, inheriting its expressive power and flexibility. The system structure, however, is no longer a uniform hierarchy of components. Instead, ti consists of instances of types of components organized in hierarchies of *motifs* (see figure 15). Each motif defines how hosted components interact and reconfigure via *coordination rules*. Thus, a given type of component can be subject to different rules when it is in a "ring" motif or in a "pipeline" one. Motifs themselves are typed, and each motif instance can "migrate" from one to another, dynamically changing their hierarchy

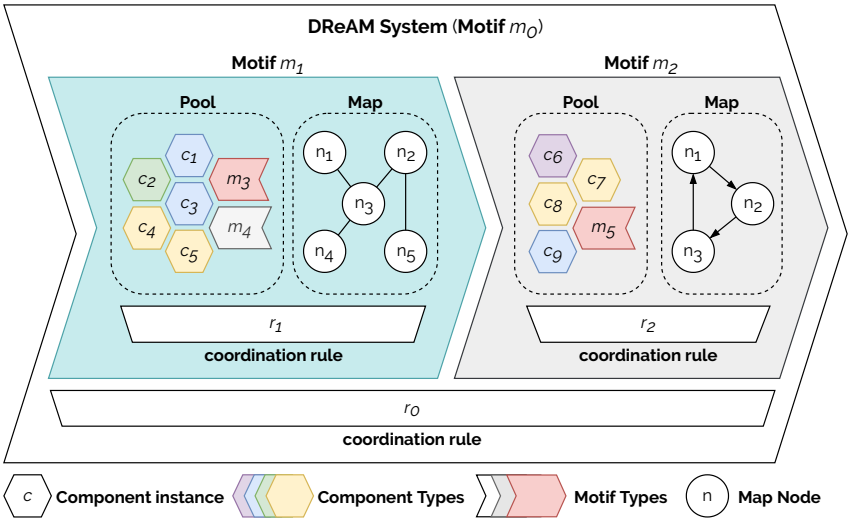


Figure 15: Overview of a DReAM system

and thus potentially affecting all the underlying components. Coordination rules in a motif are expressed using the same formalism adopted for L-DReAM rules in (3.33). Component instances can also migrate between motifs, allowing to switch seamlessly coordination style at runtime.

To enhance expressiveness of the different kinds of dynamism, each motif is equipped with a *map*, which is a graph defining the topology of the interactions in the motif. To parametrize coordination terms for the nodes of the map, an addressing function @ is provided which defines the position $@(c)$ in the map of any component instance c associated with the motif. Additionally, each node is equipped with a local memory that can be accessed by components and used as a shared memory. Maps are also very useful to express mobility, in which case the connectivity relation of the map represents possible moves of components. Finally, the language allows to modify maps by adding or removing nodes and edges, as well as to dynamically create and delete component instances.

4.1 Structuring architectures

A DReAM system is made of atomic components, each one characterized by a behavior, a data store for local variables, and an interface of ports to interact and coordinate with other components. The system architecture that realizes their coordination is structured with hierarchies of "motifs", each representing an independent architecture with its own coordination rules and data structures for parametrization.

The support for dynamic reconfiguration allows systems to modify their architecture at runtime by creating, deleting, and migrating both components and motifs. This allows to model changes in the population of entities in the system but also seamless switching of architectural styles.

In this section we will introduce and formally define the entities that characterize the structure of a DReAM system.

4.1.1 Component Types and component Instances

The basic elements of DReAM systems are *instances of component types*. Component types in DReAM are essentially a mix between L-DReAM component types (Definition 3.2.3) and basic interacting components (Definition 3.1.1).

Concretely, DReAM component types are labeled transition systems equipped with a *store* (i.e., a set of *local variables*) and an *interface* (i.e., a set of *ports*). Transitions connect *control locations* of the LTS, and their labels are *transition rules* built from a fragment of PILOps (P, X) that we will call t-PILOps:

$$\begin{aligned} &\text{(t-PILOps rule)} && t ::= \Phi \to \Delta \\ &\text{(PIL atom)} && \Phi ::= idle(P) \mid p \in P \\ &\text{(operation set)} && \Delta ::= \mathbf{skip} \mid \{x := f(X)\} \mid \Delta_1 \cup \Delta_2 \end{aligned} \tag{4.1}$$

where:

- $idle(P) \equiv \bigwedge_{p \in P} \neg p$ is a notation to represent the PIL monomial obtained by the conjunction of all the negated ports in P;
- $x \in X$ is a local variable.

For the sake of conciseness, we will omit the the operation part of the transition rule when $\Delta = \mathbf{skip}$ (i.e., $\Phi \equiv \Phi \rightarrow \mathbf{skip}$).

The models of the fragment and semantics of operations are the same as static PILOps defined in section 3.2.1, with the difference that the effect of the application of operations to components states $\Delta(\gamma)$ is limited to the valuation function of the component (as DReAM components do not host other components like L-DReAM compounds, and operation sets associated to transition only contain assignments).

Having introduced t-PILOps, we can now formally define what a component type is in DReAM.

Definition 4.1.1 (DReAM Component Type)**.** Let $\mathcal{S}$ be the set of all component control locations, $\mathcal{X}$ the set of all local variables, and $\mathcal{P}$ the set of all ports. A *component type* ct is a transition system $ct := (P, X, S, T, \gamma_0)$, where:

- *interface* $P \subseteq \mathcal{P}$: finite set of ports;
- *store* $X \subseteq \mathcal{X}$: finite set of local variables;
- $S \subseteq \mathcal{S}$: finite set of control locations;
- $T \subseteq S \times \text{t-PILOps}(P, X) \times S$: finite set of transitions $\langle s, t, s' \rangle$, where $t \in \text{t-PILOps}(P, X)$;
- *initial state* $\gamma_0 \in \Gamma$ of the form $\gamma_0 = (s_0, \sigma_0)$, with $s_0 \in S$ and $\sigma_0 : X \mapsto \mathsf{V}$ is a valuation function for local variables in X.

Each component type has one implicit loop transition $\langle s, idle(P), s \rangle$ for each control location $s \in S$. This transition represents the *inaction*, i.e., the ability of instances of component type ct of not participating in interactions by not changing their state. For the sake of simplicity, we assume that no transition other than the idle ones are labeled with the PIL formula "$idle(P)$" (i.e., every non-idle transition is labeled with a port name).

Just like in L-DReAM, component instances are obtained from a component type by renaming its control locations, ports and local variables with a unique *identifier*.

Definition 4.1.2 (DReAM Component Instance)**.** Let Cid be the domain of *component identifiers* and $ct = (P, X, S, T, \gamma_0)$ be a component type.

A *component instance* c of type ct, identified by $i \in \mathsf{Cid}$, is obtained by renaming the set of control locations, ports and local variables of the component type ct with the identifier i, that is:

$$c \triangleq ct[i] = (P[i], X[i], S[i], T[i], \gamma_0[i]) \tag{4.2}$$

where:

- the initial state $\gamma_0[i]$ is obtained from $\gamma_0 = (s_0, \sigma_0)$ by replacing the initial control location s_0 with the renamed $s_0[i] \in S[i]$ and by replacing the domain of σ_0 with $X[i]$;
- transitions $T[i]$ are obtained from T by replacing control locations, ports and local variables with their renamed instances:

$$T[i] = \big\{ \langle s[i], t[i], s'[i] \rangle \mid \\ \langle s, t, s' \rangle \in T,\ t[i] = t\,[p[i]/p]\,[x[i]/x]\ \ \forall p \in P, \forall x \in X \big\}$$

Without loss of generality, we assume that instance identifiers uniquely represent a component instance regardless of its type.

The state γ_c of a component instance c is therefore defined as the pair $\gamma_c = (c.s, c.\sigma)$, where $c.\sigma$ is the *valuation function* of the variables $c.X$. Σ denotes the domain of all valuation functions, while we can refer to the set of all possible valuation functions for component instance c with $c.\Sigma$.

We use the same notation to denote ports, control locations, states and variables belonging to a given component instance (e.g. $c.p \in c.P$). Given that, by construction, each instance identifier is uniquely associated to one component instance, we have that sets of ports, control location and local variables of different component instances are disjoint, i.e. $c.P \cap c'.P = \emptyset$ for $c \neq c'$.

From their "internal" behavioral standpoint, component instances are essentially interacting components (Def. 3.1.1) with transitions possibly enriched with sets of assignments. As a result, the operational semantics of interactions for component instances is defined according to the following rule:

$$\frac{a \models \Phi \to \Delta \qquad \langle c.s, \Phi \to \Delta, c.s' \rangle \in c.T \qquad c.\sigma' = \Delta\,(c.\sigma)}{(c.s, c.\sigma) \xrightarrow{a} (c.s', c.\sigma')} \tag{4.3}$$

It is worth pointing out that the hierarchical structuring of L-DReAM components does not apply here, meaning that all DReAM component instances are in fact L-DReAM atomic components. The role of compounds as aggregates of atomic components defining how they interact, coordinate and reconfigure, is instead embodied by the specialized concept of *motif*.

Example 4.1.1 (Client-Server architecture with back-end nodes: component types)**.** Recall the scenario presented in chapter 3 by examples 3.2.1-3.2.4 where we defined a portion of a client-server architecture with $Server$ components receiving data fed to $Node$ components to process and then return the result of the computation. Here we are going to model the same general scenario using the DReAM framework. Instead of focusing only on the "server-side" of the system, now we also model the *clients* querying the service offered by the servers that we defined in the simple PIL scenario of example 3.1.1.

We start by re-defining the component types $Client$, $Server$ and $Node$.

$Client$ **component type:** clients are simple two-stages automatons that $post\ data$ to servers and then get the $result$ back once ready (figure 16). The operation Δ_{get} associated to the transition from $wait$ to $ready$ handles the consumption of the $result$ from the service and generates new $data$. Additionally, each client has a local variable $server$ to store the identifier of the server queried.

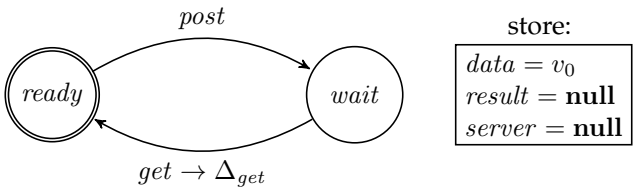


Figure 16: The $Client$ component type

$Server$ **component type:** servers $accept$ requests and $return$ results from $Client$ components, and handle the computation in a distributed fashion by *mapping* input data to $Node$ instances and *reducing* the individual results to a single $output$ with operation Δ_{red} (figure 17). The $buffer$ vari-

able is an array used to store individual results returned from nodes (i.e. $buffer : \mathsf{V} \mapsto \mathsf{V}$). Each server keeps also track of the identifier of the client that requested the current computation in the local variable $client$.

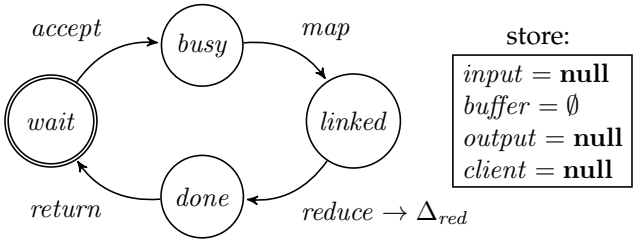


Figure 17: The $Server$ component type

$Node$ **component type:** nodes are modeled with a two-stage automaton that can $receive$ new $data$ to process, and can $return$ the computation $result$ when done (figure 18). To ensure that the result is forwarded to the correct server instance (if more than one is present), each node also stores the identifier of the server that requested the computation in the local variable $server$.

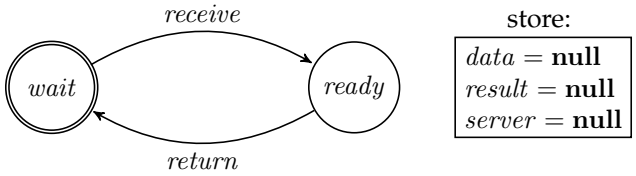


Figure 18: The $Node$ component type

4.1.2 Motif modeling

A motif characterizes an independent dynamic architecture defined by a *coordination rule* and parametrized by a data structure called *map*. Just like component types serve as blueprints for actual executable component instances, *motif type* definitions can be used to create executable *motif instances*.

Definition 4.1.3 (Motif Type)**.** A *motif type* mt is a pair $mt := (r, \mu_0)$, where r is the coordination rule characterizing the interaction and reconfiguration architecture of mt, and μ_0 is the initial configuration of the map associated to the motif.

A motif can host and coordinate component instances as well as other nested motifs. We refer to the combined set of motifs and components hosted in another motif as its *pool*. Every entity hosted in a motif is also associated to its map. A *map* is a set of *locations* and a connectivity relation between them. It is the structure over which computation is distributed and defines a system of coordinates for hosted components and motifs, which are correlated to locations via an *address function*. It can represent a physical structure, such as a geographic map, or some conceptual structure, e.g., the cellular structure of a memory. Additionally, each location has a local memory that can written to or read from.

Formally, a map in DReAM is a pair $(\mu_0, \mathcal{F})$, where μ_0 is its initial configuration and $\mathcal{F}$ is a set of utility functions (e.g. predicates on reachability from one location to another, expressions that return the number of edges connected to a location, etc). Having $\mathcal{M}$ and $\mathcal{C}$ respectively as the domain of motifs and component instances, the configuration μ of a map in DReAM is specified as a graph $\mu = \big(N, E, \omega, @\big)$, where:

- N is a set of nodes or locations (possibly infinite);
- E is a set of (possibly directed) edges subset of $N \times N$ that defines the connectivity relation between nodes;
- $\omega : N \mapsto \mathsf{V}$ is a valuation function that associates nodes to values, realizing the map memory;
- $@ : \mathcal{M} \cup \mathcal{C} \to N$ is a (partial) address function binding motifs and components to nodes $n \in N$.

If the map memory is empty, then the only available information for a location is its name. Otherwise, the memory can be shared by different entities and used for their coordination.

The relation E defines a concept of neighborhood, which is used in many applications to express coordination constraints or directions for

moving components. When these additional topological relations are not needed and $E = \emptyset$, the map can be still used as a simple indexing structure.

As we mentioned, maps can be used to model a physical environment where components are moving. For example, we can model a discrete two-dimensional space with an array map $N = \{(i, j) \mid i, j \in \mathbb{N}\} \times \{f, o\}$, where the pairs (i, j) represent coordinates and the symbols f and o stand respectively for free and obstacle. We can model the movement of c, such that $@(c) = ((i, j), f)$, to a position $(i + a, j + b)$ provided that there is a path from (i, j) to $(i + a, j + b)$ consisting of free locations.

A map is also equipped with a set of useful functions $\mathcal{F}$. These may range from predicates over the map nodes and edges (e.g., a predicate that evaluates to **true** if there exists a sequence of edges connecting two nodes), to selectors that return a specific location having a specific property (e.g., a function that evaluates to the node with the highest number of adjacent edges).

Definition 4.1.4 (Motif Instance)**.** Let $mt = (r, \mu_0)$ be a motif type and Mid the domain of *motif identifiers*. A *motif instance* $mt[i]$ is obtained from mt by renaming the constituent elements of the initial map configuration $mt.\mu_0$ with the identifier $i \in$ Mid and all their occurrences in the coordination rule r:

$$m \triangleq mt[i] = (r\left[\mu_0[i]/\mu_0\right], \mu[i]) \tag{4.4}$$

where $\mu_0[i] = (N[i], E\left[N[i]/N\right], \omega\left[N[i]/N\right], @\left[N[i]/N\right])$.

The state γ_m of a motif instance m is the combination of a map configuration μ with the sets of hosted motifs M, component instances C, and respective current states:

$$\gamma_m = (M, C, \Gamma_M, \Gamma_C, \mu) \tag{4.5}$$

where $\Gamma_M = \{\gamma_{m_i}\}_{m_i \in M}$ and $\Gamma_C = \{\gamma_{c_i}\}_{c_i \in C}$.

We assume that each motif and component instance is hosted in exactly one motif instance, i.e. $\gamma_{m_1}.M \cap \gamma_{m_2}.M = \emptyset$ and $\gamma_{m_1}.C \cap \gamma_{m_2}.C = \emptyset$ for any $m_1 \neq m_2$.

Like with components, we assume that motif identifiers uniquely single out motif instances regardless of their type.

By modifying the state of a motif instance we can model:

- *Component dynamism*: the set of component instances C hosted in the motif (or in any descendant motif) may change by creating/deleting or migrating components and motifs;

- *Map dynamism*: the set of nodes or/and the connectivity relation of a map may change. This is the case in particular when an autonomous component e.g. a robot, explores an unknown environment and builds a model of it, or when nodes in a network become unreachable;

- *Mobility dynamism*: the address function $\mu.@$ changes to express mobility of components and groups of components in a motif.

Different types of dynamism can be obtained as the combination of these three basic types. More details on how they can be implemented in DReAM will be discussed in section 4.2.2 where *reconfiguration operations* are introduced.

Example 4.1.2 (Client-Server architecture with back-end nodes: motif types)**.** Let us continue example 4.1.1 by defining the motif types that will host components in the system. More specifically, we are going to model the described scenario using two motif types: the *Service* and the *ClientService* types.

Each instance of the *Service* motif type represents one independent sub-system of servers and nodes. The *ClientService* motif type defines instead the context in which clients can interact with *Service* instances, and in this case coincides with the whole system. We consider the case of a *ClientService* system initially hosting two *Client* instances and two *Service* instances, while *Service* instances start with one *Server* and two *Node* instances.

Let us define the maps associated with each motif type. For the sake of simplicity, we will assume that servers and nodes implementing services are fully connected. This means that the *Service* motif type does not require any explicit notion of topology, and its associated map can be a single location to which all components are mapped.

For the *ClientService* motif type, we want to model the (relative) geographical distribution of clients and services. To do so, we use as map

for the motif a graph with N locations, to which *Client* and *Service* instances are assigned, linked by edges that represent the notion of topological proximity in the connectivity network. For the initial configuration $\mu_0 = (N, E, \omega, @)$ we have:

$$\mu_0 = (\{n_0, \ldots, n_3\}, \{(n_0, n_2), (n_1, n_2), (n_1, n_3)\}, \emptyset, @_0)$$

that is N is a set of 4 locations and E is a set of 3 edges connecting them as in figure 19. The initial address function $@_0$ maps clients and services in such a way that each client's location is linked with at least one location mapped to a service:

$$@_0 = \{Client_1 \mapsto n_0, Client_2 \mapsto n_1, Service_1 \mapsto n_2, Service_2 \mapsto n_3\}$$

The support functions $\mathcal{F}$ of the map only includes a predicate

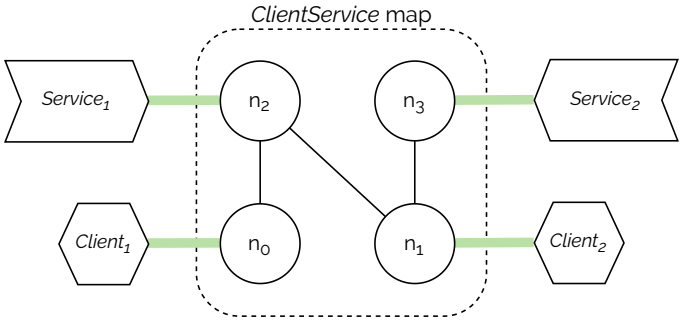


Figure 19: The initial configuration of the map of *ClientService* motif type

isEdge(n_1, n_2) that evaluates to **true** only if $(n_1, n_2) \in E$. This predicate will allow us to test "reachability" of services by clients when we define the coordination rules of the *ClientService* motif type.

4.2 The DReAM coordination language

The coordination language modeling interactions between components is, for the most part, independent from the abstract system structure. Consequently, the coordination language introduced in section 3.2.3 is shared between L-DReAM and DReAM.

Given the domain of ports $\mathcal{P}$ and the set of all possible system states Γ, the DReAM coordination language is therefore obtained from (3.33) by replacing the generic compound c^* with the motif m in the scope of the declarations:

$$\begin{aligned}
\text{(DReAM rule)} \quad & r ::= \Psi \to \Delta \mid D\{r\} \mid r_1 \,\&\, r_2 \mid r_1 \parallel r_2 \\
\text{(declaration)} \quad & D ::= \forall m : m_s.mt \mid \exists m : m_s.mt \mid \forall c : m_s.ct \mid \exists c : m_s.ct \\
\text{(PIL formula)} \quad & \Psi ::= \mathbf{true} \mid p \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \\
\text{(operation set)} \quad & \Delta ::= \mathbf{skip} \mid \{\delta\} \mid \Delta_1 \cup \Delta_2
\end{aligned} \tag{4.6}$$

where *declarations* define the context of the associated rule by declaring quantified ($\forall|\exists$) variables for motifs (m) and components (c) associated to instances of a given type (mt and ct, respectively) hosted in a "scope" motif m_s.

Note that predicates π now include map functions in $\mathcal{F}$ with Boolean codomain (e.g., the $\mathsf{isEdge}(n_1, n_2)$ predicate used in example 4.1.2) and can use other map functions in equality/inequality relations (e.g., having defined the function $\mathsf{tail}(m)$ that returns the last location in a map modeling an ordered sequence of nodes, we can define a predicate that is satisfied when the component c is located in this last location as $@(c) = \mathsf{tail}(m)$).

A DReAM coordination rule is *well formed* if all motifs and component variables appearing in the scope of declarations, PIL formulas and associated operations, are defined in a declaration. From now on, we will only consider well formed terms.

Just like in L-DReAM, given a system state, a DReAM rule can be translated to an equivalent one by performing a *declaration expansion* step, which expands the quantifiers and replaces variables with actual motif and component instances.

4.2.1 Declaration expansion for coordination terms

Given that DReAM systems host a finite number of motif and component instances, first-order logic quantifiers can be eliminated by enumerating every one of them according to the type specified in each declaration. We

thus define the *declaration expansion* $\langle\langle r \rangle\rangle_{\gamma_{\hat{m}}}$ of rule r in motif $\hat{m}$ under state $\gamma_{\hat{m}}$ by adapting rules (3.34) as follows:

$$
\begin{aligned}
\langle\langle \Psi \to \Delta \rangle\rangle_{\gamma_{\hat{m}}} &= \Psi \to \Delta \\
\langle\langle \forall m : m_s . mt\{r\} \rangle\rangle_{\gamma_{\hat{m}}} &= \mathop{\&}_{mt[i] \in \gamma_{m_s}.M} \langle\langle r\,[ct[i]/c] \rangle\rangle_{\gamma_{\hat{m}}} \\
\langle\langle \exists m : m_s . mt\{r\} \rangle\rangle_{\gamma_{\hat{m}}} &= \mathop{\Big\|}_{mt[i] \in \gamma_{m_s}.M} \langle\langle r\,[mt[i]/m] \rangle\rangle_{\gamma_{\hat{m}}} \\
\langle\langle \forall c : m_s . ct\{r\} \rangle\rangle_{\gamma_{\hat{m}}} &= \mathop{\&}_{ct[i] \in \gamma_{m_s}.C} \langle\langle r\,[ct[i]/c] \rangle\rangle_{\gamma_{\hat{m}}} \\
\langle\langle \exists c : m_s . ct\{r\} \rangle\rangle_{\gamma_{\hat{m}}} &= \mathop{\Big\|}_{ct[i] \in \gamma_{m_s}.C} \langle\langle r\,[ct[i]/c] \rangle\rangle_{\gamma_{\hat{m}}} \\
\langle\langle r_1 \,\&\, r_2 \rangle\rangle_{\gamma_{\hat{m}}} &= \langle\langle r_1 \rangle\rangle_{\gamma_{\hat{m}}} \,\&\, \langle\langle r_2 \rangle\rangle_{\gamma_{\hat{m}}} \\
\langle\langle r_1 \parallel r_2 \rangle\rangle_{\gamma_{\hat{m}}} &= \langle\langle r_1 \rangle\rangle_{\gamma_{\hat{m}}} \parallel \langle\langle r_2 \rangle\rangle_{\gamma_{\hat{m}}}
\end{aligned}
\tag{4.7}
$$

where:

- m_s is either a descendant of $\hat{m}$ or they coincide;
- $r\,[mt[i]/m]$ is the substitution of the motif variable m with the actual instance $mt[i]$ in rule r;
- $r\,[ct[i]/c]$ is the substitution of the component variable c with the actual instance $ct[i]$ in rule r.

For the sake of conciseness, (4.6) and (4.7) do not mention additional notations that can be used to ease declaration writing. For instance, explicit reference to the motif hosting the instance variable can be omitted when the term is associated to the motif itself (otherwise referenced through the `this` keyword). Similarly, if the rule has to apply to groups of instances regardless of their type, the component type can be omitted in the declaration.

By applying (4.7) exhaustively under a given system state, any rule can be transformed into an equivalent one only involving actual motifs and component instances free of declarations. Notice that this transformation is only practically feasible at runtime when the exact hierarchy of component and motif instances is known.

4.2.2 Reconfiguration operations

In addition to the "assignment" operation already introduced for PILOps in section 3.2.1, DReAM supports more complex reconfiguration operations which enable component, map and mobility dynamism by allowing transformations of a motif configuration at run-time. Some of them are natural extensions of the ones introduced for L-DReAM in section 3.2.3, while others are entirely new and exclusive to DReAM.

We will now list all the main operations that enable reconfiguration capabilities breaking them down into three classes of dynamism:

1. Motif dynamism can be realized using the following statements:
   - create$(m : mt, m_s, n, g)\ [\Delta_m]$: to create a new motif instance of type mt at node n of the map of motif m_s, binding it to the variable m within the scope of the operation Δ_m, which is then executed;
   - delete(m): to delete motif instance m (and all its descendants).
2. Component dynamism can be realized using the following statements:
   - create$(c : ct, m_s, n, g)\ [\Delta_c]$: to create a component instance of type ct at node n of the map of motif m_s, binding it to the variable c within the scope of the operation Δ_c, which is then executed;
   - delete(c): to delete component instance c.
3. Map dynamism can be realized using the following statements:
   - addNode$(n, m_s, g)\ [\Delta_n]$: to create a new location in the configuration of the map of motif m_s, binding it to variable n within the scope of the operation Δ_n, which is then executed;
   - rmNode(n, m_s): to remove node n from the configuration of the map of motif m_s, along with incident edges, mapped motifs and components;

- addEdge(n_1, n_2, m_s): to add the edge (n_1, n_2) to the configuration of the map of motif m_s;
- rmEdge(n_1, n_2, m_s): to remove the edge (n_1, n_2) from the configuration of the map of motif m_s.

4. Mobility dynamism can be realized using the following statements:
   - move(m, n): to change the position of motif instance m to node n in the map of motif m_s;
   - move(c, n): to change the position of component instance c to node n in the map of motif m_s.
5. Dynamic changes of architectural constraints can be realized with the following statements:
   - migrate(m, m_s, n): to move a motif instance m to node n in the map of the motif instance m_s;
   - migrate(c, m_s, n): to move a component instance c to node n in the map of the motif instance m_s.

where g is a *generator* attribute that is used to discriminate multiple operations with the same parameters as in the L-DReAM "create" operation (described in section 3.2.3).

Semantics of reconfiguration operations

The application of the listed reconfiguration operations on the state of a motif produces a new state. Let $\hat{m}$ be a motif and $\hat{m}.\gamma = (M, C, \Gamma_M, \Gamma_C, \mu)$ be its state, where $\mu = \big(N, E, \omega, @\big)$ is the configuration of the associated map. To formalize the transformation induced by the application of each different reconfiguration operation on the state $\hat{m}.\gamma$, let us first define the semantics of the application of the support "binding" operations as follows:

$$\text{bind}\,(m, m_s, n)\,(\hat{m}.\gamma) = \begin{cases} (M \oplus m, C, \Gamma_M \oplus m.\gamma_0, \Gamma_C, \mu') & \text{if } m_s = \hat{m} \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.8}$$

where:

- $\mu' = \big(N, E, \omega, @\,[m \mapsto n]\,\big)$;
- $\Gamma'_M = \{\mathsf{bind}\,(m, m_s, n)\,(\gamma_i)\ \mid \gamma_i \in \Gamma_M\}$;
- $S \oplus s \equiv S \cup \{s\}$ is the set addition operation.

$$\mathsf{bind}\,(c, m_s, n)\,(\hat{m}.\gamma) = \begin{cases} (M, C \oplus c, \Gamma_M, \Gamma_C \oplus c.\gamma_0, \mu') & \text{if } m_s = \hat{m} \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.9}$$

where:

- $\mu' = \big(N, E, \omega, @\,[c \mapsto n]\,\big)$;
- $\Gamma'_M = \{\mathsf{bind}\,(c, m_s, n)\,(\gamma_i)\ \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{bindNode}(n, m_s)\,(\hat{m}.\gamma) = \begin{cases} (M, C, \Gamma_M, \Gamma_C, \mu') & \text{if } m_s = \hat{m} \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.10}$$

where:

- $\mu' = \big(N \oplus n, E, \omega, @\big)$;
- $\Gamma'_M = \{\mathsf{bindNode}(n, m_s)\,(\gamma_i)\ \mid \gamma_i \in \Gamma_M\}$.

As already mentioned for L-DReAM, the bind and bindNode operations are introduced to simplify the presentation of the semantics of more complex reconfiguration operations, which we will now detail.

$$\mathsf{create}(m : mt, m_s, n, g)\,\big[\Delta_m\big]\,(\hat{m}.\gamma) = (\Delta_m\,[mt[f]\,/m])\!\downarrow_{\gamma'_{\hat{m}}}(\gamma'_{\hat{m}}) \tag{4.11}$$

where:

- $f \in \mathsf{Mid}$ is a *fresh* motif instance identifier;
- $\gamma'_{\hat{m}} = \mathsf{bind}\,(mt[f]\,, m_s, n)\,(\hat{m}.\gamma)$.

$$\mathsf{delete}(m)\,(\hat{m}.\gamma) = \begin{cases} (M \ominus m, C, \Gamma_M \ominus m.\gamma, \Gamma_C, \mu') & \text{if } m \in M \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.12}$$

where:

- $\mu' = \big(N, E, \omega, @\,[m \mapsto \bot]\,\big)$, i.e., remove m from the domain of $@$;
- $\Gamma'_M = \{\mathsf{delete}(m)\,(\gamma_i) \mid \gamma_i \in \Gamma_M\}$;
- $S \ominus s \equiv S \setminus \{s\}$ is the set removal operation.

$$\mathsf{create}(c\!:\!ct, m_s, n, g)\,\big[\Delta_c\big]\,(\hat{m}.\gamma) = (\Delta_c\,[ct[f]\,/c])\!\downarrow_{\gamma'_{\hat{m}}}(\gamma'_{\hat{m}}) \tag{4.13}$$

where:

- $f \in \mathsf{Cid}$ is a *fresh* component instance identifier;
- $\gamma'_{\hat{m}} = \mathsf{bind}\,(ct[f]\,, m_s, n)\,(\hat{m}.\gamma)$.

$$\mathsf{delete}(c)\,(\hat{m}.\gamma) = \begin{cases} (M, C \ominus c, \Gamma_M, \Gamma_C \ominus c.\gamma, \mu') & \text{if } C \in C \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.14}$$

where:

- $\mu' = \big(N, E, \omega, @\,[c \mapsto \bot]\,\big)$, i.e., remove c from the domain of $@$;
- $\Gamma'_M = \{\mathsf{delete}(c)\,(\gamma_i) \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{addNode}(n, m_s, g)\,\big[\Delta_n\big]\,(\hat{m}.\gamma) = (\Delta_n\,[n'/n])\!\downarrow_{\gamma'_{\hat{m}}}(\gamma'_{\hat{m}}) \tag{4.15}$$

where:

- n' is a new map location;
- $\gamma'_{\hat{m}} = \mathsf{bindNode}(n', m_s)\,(\hat{m}.\gamma)$.

$$\mathsf{rmNode}(n, m_s)\,(\hat{m}.\gamma) = \begin{cases} (M', C', \Gamma_{M'}, \Gamma_{C'}, \mu') & \text{if } m_s = \hat{m} \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.16}$$

where:

- the map configuration μ' is obtained from μ by: removing node n, removing all edges incident to n, removing the valuation for n from ω, and keeping only the the mappings towards location different from n in the addressing function $@$:

$$\mu' = \big(N \ominus n, \{(n_1, n_2) \mid (n_1, n_2) \in E : n_1 \neq n \wedge n_2 \neq n\}, \\ \omega\,[n \mapsto \bot], \{(e \mapsto n_i) \mid (e \mapsto n_i) \in @ : n_i \neq n\}\big)$$

- $M' = M \setminus \{m \mid m \in M : @(m) = n\}$;
- $C' = C \setminus \{c \mid c \in C : @(c) = n\}$;
- $\Gamma'_M = \{\mathsf{rmNode}(n, m_s)\,(\gamma_i) \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{addEdge}(n_1, n_2, m_s)\,(\hat{m}.\gamma) = \begin{cases} (M, C, \Gamma_M, \Gamma_C, \mu') & \text{if } m_s = \hat{m} \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.17}$$

where:

- $\mu' = \big(N, E \oplus (n_1, n_2), \omega, @\big)$ if $n_1, n_2 \in N$, otherwise $\mu' = \mu$;
- $\Gamma'_M = \{\mathsf{addEdge}(n_1, n_2, m_s)\,(\gamma_i) \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{rmEdge}(n_1, n_2, m_s)\,(\hat{m}.\gamma) = \begin{cases} (M, C, \Gamma_M, \Gamma_C, \mu') & \text{if } m_s = \hat{m} \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.18}$$

where:

- $\mu' = \big(N, E \ominus (n_1, n_2), \omega, @\big)$ if $n_1, n_2 \in N$, otherwise $\mu' = \mu$;
- $\Gamma'_M = \{\mathsf{rmEdge}(n_1, n_2, m_s)\,(\gamma_i) \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{move}(m, n)\,(\hat{m}.\gamma) = \begin{cases} (M, C, \Gamma_M, \Gamma_C, \mu') & \text{if } m \in M \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.19}$$

where:

- $\mu' = \left(N, E, \omega, @\left[m \mapsto n\right]\right)$ if $n \in N$, otherwise $\mu' = \mu$;
- $\Gamma'_M = \{\mathsf{move}(m, n)\left(\gamma_i\right) \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{move}(c, n)\left(\hat{m}.\gamma\right) = \begin{cases} (M, C, \Gamma_M, \Gamma_C, \mu') & \text{if } c \in C \\ (M, C, \Gamma'_M, \Gamma_C, \mu) & \text{otherwise} \end{cases} \tag{4.20}$$

where:

- $\mu' = \left(N, E, \omega, @\left[c \mapsto n\right]\right)$ if $n \in N$, otherwise $\mu' = \mu$;
- $\Gamma'_M = \{\mathsf{move}(c, n)\left(\gamma_i\right) \mid \gamma_i \in \Gamma_M\}$.

$$\mathsf{migrate}(m, m_s, n)\left(\hat{m}.\gamma\right) = \mathsf{bind}\left(m, m_s, n\right)\left(\mathsf{delete}(m)\left(\hat{m}.\gamma\right)\right) \tag{4.21}$$

$$\mathsf{migrate}(c, m_s, n)\left(\hat{m}.\gamma\right) = \mathsf{bind}\left(c, m_s, n\right)\left(\mathsf{delete}(c)\left(\hat{m}.\gamma\right)\right) \tag{4.22}$$

4.2.3 Disjunctive and Conjunctive styles in DReAM

As we have seen, DReAM inherits the core coordination language to define its rules from L-DReAM. As such, the two frameworks naturally share also the different approaches on system specification that we called *disjunctive* and *conjunctive* presented in sections 3.1.4 and 3.2.2. Notably, the coordination language defined in (4.6) is enriched with the same *conjunctive term* formalized in definition 3.2.2.

The notion of "consistency", introduced for conjunctive PILOps terms, is also extended to encompass the richer set of reconfiguration operations and the different system structure. That is, having ρ_c be either a port $c.p$ or a predicate $\pi\left(c.X\right)$, we consider a conjunctive term $\rho_c \rhd \Psi_{\rho_c} \rightarrow \Delta_{\rho_c}$ to be *consistent* if its consequence Δ_{ρ_c} is equal to $\mathbf{skip}$, or if it only contains operations δ that do not modify the state of components different from c nor their addressing in the relevant map. Concretely, this allows Δ_{ρ_c} to include any operation that modifies the state of c and of its ancestors as long as it does not delete, move, or migrate another component instance, nor it performs an assignment on one of its local variables.

A DReAM rule for motif $\hat{m}$ is written using the conjunctive style by combining consistent conjunctive terms with the $\&$ operator.

Example 4.2.1 (Client-Server architecture with back-end nodes: coordination rules)**.** In example 4.1.2 we described the general architecture of the system and the initial configuration of the map for motif type *ClientService*. Now we define the coordination rules that realize the interaction patterns described qualitatively in example 4.1.1 adopting the conjunctive style.

Let us start with rule r_{cs} of the *ClientService* motif type. The interactions that need to be modeled are of two kinds, both involving a *Client* instance and a *Server* instance: one that starts a task on the client's data, and one that transfers the results of the task back to the client. We can separate r_{cs} into two sub-rules each modeling the two kinds of interactions listed:

- rule r^1_{cs}, which models the strong synchronization between clients and servers that are mapped to connected locations in the map allowing the former to send data to process to the latter:

$$
\begin{aligned}
r^1_{cs} = & \forall c : \mathit{Client}\big\{\exists s : \mathit{Service.Server}\big\{ \\
& \quad c.\mathit{post} \rhd s.\mathit{accept} \wedge \mathsf{isEdge}\big(@(c)\,, @(s)\,\big) \\
& \quad \rightarrow \{c.\mathit{server} := s.\mathit{id}\}\,\big\}\big\} \\
& \& \\
& \forall s : \mathit{Service.Server}\big\{\exists c : \mathit{Client}\big\{ \\
& \quad s.\mathit{accept} \rhd c.\mathit{post} \wedge \mathsf{isEdge}\big(@(c)\,, @(s)\,\big) \\
& \quad \rightarrow \{s.\mathit{client} := c.\mathit{id}, s.\mathit{input} := c.\mathit{data}\}\,\big\}\big\}
\end{aligned}
$$

- rule r^2_{cs}, which models the strong synchronization between clients and servers that are mapped to connected locations in the map allowing the former to receive the results of the computation from the latter:

$$
\begin{aligned}
r^2_{cs} = & \forall c : \mathit{Client}\big\{\exists s : \mathit{Service.Server}\big\{ \\
& \quad c.\mathit{get} \rhd s.\mathit{return} \wedge \mathsf{isEdge}\big(@(c)\,, @(s)\,\big) \wedge (s.\mathit{id} = c.\mathit{server}) \\
& \quad \rightarrow \{c.\mathit{result} := s.\mathit{output}\}\,\big\}\big\} \\
& \& \\
& \forall s : \mathit{Service.Server}\big\{\exists c : \mathit{Client}\big\{ \\
& \quad s.\mathit{return} \rhd c.\mathit{get} \wedge \mathsf{isEdge}\big(@(c)\,, @(s)\,\big) \wedge (c.\mathit{id} = s.\mathit{client}) \\
& \quad \rightarrow \mathbf{skip}\big\}\big\}
\end{aligned}
$$

Notice that in the declaration of the component variable s we used a compressed notation that implicitly defines the scope of the variable in any *Service* motif instance:

$$\forall c : mt.ct\{r\} \equiv \forall m : mt\{\forall c : m.ct\{r\}\}$$
$$\exists c : mt.ct\{r\} \equiv \exists m : mt\{\exists c : m.ct\{r\}\}$$

To ensure that the initial synchronization between a client and a server does not involve more than one instance of each, we also add a rule of the form:

$$r^3_{cs} = \mathsf{atMost}(1, \mathit{Client.post}) \ \& \ \mathsf{atMost}(1, \mathit{Service.Server.accept})$$

where $\mathsf{atMost}(k, mt.ct.p)$ is a macro notation for a rule that is satisfied by interactions in which there are at most k component instances of type ct in a motif of type mt participating with port p:

$$\mathsf{atMost}(k, mt.ct.p) \triangleq \forall m : mt\left\{ \forall c_0, \ldots, c_k : m.ct\left\{ c_0.p \rhd \bigvee_{i,j \in [0..k], i \neq j} c_i = c_j \vee \bigvee_{i \in [1..k]} \neg c_i.p \rightarrow \mathbf{skip} \right\}\right\} \tag{4.23}$$

By combining these three sub-rules we obtain $r_{cs} = r^1_{cs} \ \& \ r^2_{cs} \ \& \ r^3_{cs}$.

To complete the specification, we need to define the coordination rule r_s for motif type *Service*. Since the interactions involve one *Server* component instance and some *Node* component instances in a two-phase rendezvous synchronization, we can define r_s using the same "structure" we used for r_{cs}, i.e., dividing it into two sub-rules:

- rule r^1_s, which models the strong synchronization between servers and nodes, allowing the former to send data to process to the latter:

$$\begin{aligned} r^1_s = & \forall s : \mathit{Server}\{\exists n : \mathit{Node}\{ \\ & \quad s.\mathit{map} \rhd n.\mathit{receive} \rightarrow \mathbf{skip}\}\} \\ & \& \\ & \forall n : \mathit{Node}\{\exists s : \mathit{Server}\{ \\ & \quad n.\mathit{receive} \rhd s.\mathit{map} \\ & \quad \rightarrow \{n.\mathit{server} := s.\mathit{id}; n.\mathit{data} := s.\mathit{input}\}\}\} \end{aligned}$$

where, for the sake of simplicity, we have assumed that servers transfer all the *input* data to each node.

– rule r_s^2, which models the strong synchronization between servers and nodes allowing the former to receive the results of the computation from the latter:

$$\begin{aligned} r_s^2 = & \forall s : Server \{ \exists n : Node \{ \\ & \quad s.reduce \rhd n.return \rightarrow \{ s.buffer[n.id] := n.result \} \} \} \\ & \& \\ & \forall n : Node \{ \exists s : Server \{ \\ & \quad n.return \rhd s.reduce \wedge (n.server = s.id) \rightarrow \mathbf{skip} \} \} \end{aligned}$$

We apply similar restrictions as the ones in r_{cs}^3 on the number of servers allowed to participate in an interaction:

$$r_s^3 = \mathsf{atMost}(1, Server.map) \ \& \ \mathsf{atMost}(1, Server.reduce)$$

which combined with the other two sub-rules produces $r_s = r_s^1 \ \& \ r_s^2 \ \& \ r_s^3$.

4.2.4 Operational semantics

As described in 4.1.2, motifs are equipped with rules r of the coordination language which are used to compose motif and component instances assigned to them and regulate their interactions. A motif instance m can evolve from state γ_m to state γ''_m by performing a transition labeled with the interaction a and characterized by the application of the set of operations $[\![\langle\langle r \rangle\rangle_{\gamma_m}]\!]_{a,\gamma_m}$ iff $a \models \langle\langle r \rangle\rangle_{\gamma_m}$. Formally this is encoded by the following inference rule:

$$\frac{a \models_{\gamma_m} \langle\langle r \rangle\rangle_{\gamma_m} \qquad \gamma_m \xrightarrow{a} \gamma'_m \qquad \gamma''_m \in [\![\langle\langle r \rangle\rangle_{\gamma_m}]\!]_{a,\gamma_m} (\gamma'_m)}{\gamma_m \xrightarrow{a} \gamma''_m} \ m = (r, \mu_0) \tag{4.24}$$

where:

– recall that rule (4.3) defines the semantics of individual component instances; $\gamma_m \xrightarrow{a} \gamma'_m$ expresses the capability of the motif m to

evolve to state γ'_m when its hosted motif and component instances evolve with a:

$$\frac{\forall \gamma_{m_i} \in \Gamma_M : \gamma_{m_i} \overset{a}{\rightsquigarrow} \gamma'_{m_i} \qquad \forall \gamma_c \in \Gamma_C : \gamma_c \overset{a}{\rightarrow} \gamma'_c}{(M, C, \Gamma_M, \Gamma_C, \mu) \overset{a}{\rightarrow} \left(M, C, \left\{\gamma'_{m_i}\right\}_M, \left\{\gamma'_{c_i}\right\}_C, \mu\right)} \tag{4.25}$$

– $[\![\langle\langle r \rangle\rangle_{\gamma_m}]\!]_{a,\gamma_m}(\gamma'_m)$ is the set of motif configurations obtained according to (3.10) and (3.13) by applying the operations $\Delta = [\![\langle\langle r \rangle\rangle_{\gamma_m}]\!]_{a,\gamma_m}$ to the motif configuration γ'_m.

Rules (4.24-4.25) can be intuitively understood as follows: a motif m evolves through interaction a, provided it satisfies its coordination rule, if all its hosted motifs and component instances can also evolve through a making m reach an intermediate state γ'_m; as a consequence, the final state γ''_m reached by m is obtained by applying the operations in its coordination rule evaluated in its initial state to the intermediate state γ'_m.

Definition 4.2.1 (DReAM system)**.** Let T_m and T_c be respectively a set of motif types and component types. A DReAM *system* is a tuple (T_m, T_c, m_r) where m_r is the *root motif*.

The operational semantics of a DReAM system is defined by the inference rule (4.24) applied to the state γ_{m_r} of the root motif m_r.

4.3 Example systems

We will now present how some simple application scenarios can be modeled using the DReAM coordination language.

4.3.1 Coordinating flocks of interacting robots

Consider a system with N robots moving in a square grid, each one with given initial location and initial movement direction as represented in figure 20. Robots are equipped with a sensor that can detect other peers within a specific range ρ and assess their direction: when this happens, the robot changes its own direction accordingly.

We require that robots maintain a timestamp of their last interaction with another peer: when two robots are within the range of their sensors, their direction is updated with the one having the highest timestamp. For the sake of simplicity we also assume that the grid is, in fact, a torus with no borders.

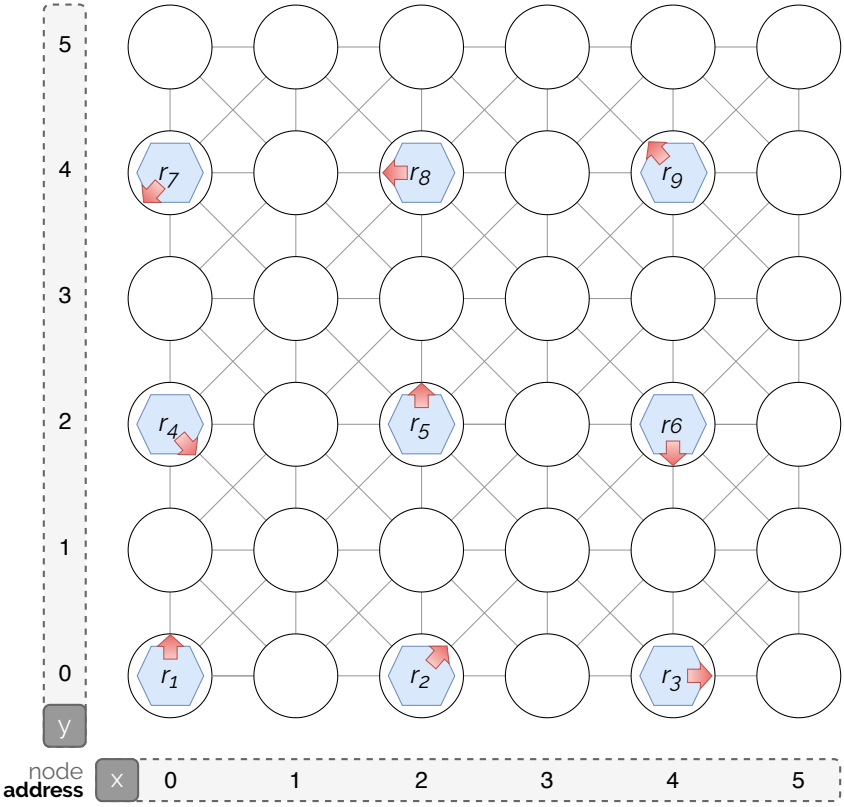


Figure 20: Initial system configuration for grid size $s = 6$

To model these robots in DReAM we will define a *Robot* component type as the one represented in figure 21. Each *Robot* maintains a local *clock* that is incremented by 1 through an assignment operation every time an instance interacts with port *tick*.

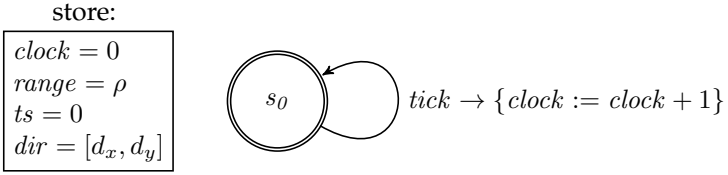


Figure 21: The *Robot* component type

A motif that realizes the described scenario can be defined with the conjunction of two coordination terms: one that enforces synchronization between every *Robot* instance through port *tick* allowing information exchange when possible, and another that enables all *Robot* instances performing a *tick* to move:

$$
\begin{aligned}
r = {} & \forall c : Robot \big\{ c.tick \rhd \mathbf{true} \rightarrow \{\mathsf{move}(c, @(c) + c.dir)\} \big\} \\
& \& \\
& \forall c_1, c_2 : Robot \big\{ c_1.tick \rhd c_2.tick \\
& \quad \rightarrow \mathsf{IF} \ \big(c_1 \neq c_2 \wedge @(c_1) \leftrightarrow @(c_2) < c_1.range \\
& \qquad \wedge (c_1.ts < c_2.ts \vee (c_1.ts = c_2.ts \wedge c_1 < c_2)) \\
& \qquad \big) \ \mathsf{THEN} \ \big\{ c_1.dir := c_2.dir, \ c_1.ts := c_1.clock \big\} \big\}
\end{aligned}
$$

where:

- we use a map whose nodes are addressed via size-two integer arrays $[x, y]$;
- we are using a $n_1 \leftrightarrow n_2$ function that returns the euclidean distance between two points in an n-dimensional space;
- in the inequality $c_1 < c_2$ we use instance variables c_1, c_2 in place of their respective integer instance identifiers.

The coordination rule r, which adopts the conjunctive style, can be intuitively understood breaking it into two parts:

1. every robot c can interact with its port $c.tick$, and if it does it also moves according to its stored direction $c.dir$;

2. for every robot c_1 to interact with its port $c_1.tick$, every robot c_2 must also participate in the interaction with its port $c_2.tick$ (i.e. interactions through port $tick$ are strictly synchronous). Furthermore, for every pair of distinct ($c_1 \neq c_2$) robots c_1, c_2 interacting through their respective $tick$ ports: if they are closer than a given range ($@(c_1) \leftrightarrow @(c_2) < c_1.range$) and either c_1 has updated its direction less recently ($c_1.ts < c_2.ts$) or they have updated their directions at the same time but c_2 has a higher instance identifier ($c_1.ts = c_2.ts \wedge c_1 < c_2$), then c_1 will update its direction and timestamp using c_2's.

Notice that if the direction of a robot is updated at a given time, the robot will move according to this new direction only during the next clock cycle because of the adopted snapshot semantics.

Since all robots synchronize on the same "clock", many of them might update their respective directions differently at the same time: adding the "tiebreaker" on the instance identifier when timestamps are equal allows data exchange even in these cases.

Multiple updates of the direction of a robot c can still happen when more than one peer with higher timestamp (or equal timestamp, but higher instance identifier) is within its range: this event does not affect the assignment on $c.ts$, which only depends on the clock, but ultimately causes the direction of c to be updated nondeterministically with one of the directions of the robots in range according to the semantics of the application of operation sets (3.13).

4.3.2 Coordinating flocks of robots with stigmergy

We consider a variant of the previous problem by using stigmergy [51]. Instead of letting robots sense each other, we will allow them to "mark" their locations with their direction and an associated timestamp. In this way, each time a robot moves to a node in the map it will either update its direction with the one stored in the node or update the one associated with the node with the direction of the robot (depending on whether the timestamp stored in the node memory is more recent than the last time

the robot changed its direction or not).

The *Robot* component type represented in figure 21 can still be used without modifications (the *range* local variable will be ignored).

The coordination term associated to the motif becomes:

$$
\begin{aligned}
r' = \forall c : Robot \big\{ & \, c.tick \\
& \rightarrow \big\{ \mathsf{IF}\ \big(@(c)\,.ts > c.ts \big)\ \mathsf{THEN}\ \big\{ \\
& \qquad c.dir := @(c)\,.dir,\ c.ts := c.clock,\ @(c)\,.ts := c.clock \\
& \big\}\ \mathsf{ELSE}\ \big\{ \\
& \qquad @(c)\,.ts := c.clock,\ @(c)\,.dir := c.dir \big\}, \\
& \mathsf{move}(c, @(c) + c.dir) \big\} \big\}
\end{aligned}
$$

Notice that we are now adopting a disjunctive-style specification for r' as the rule is made of a non-conjunctive term that does not support idling. We can interpret the term r' as follows:

1. every robot c must participate in all interactions with its port $c.tick$, and will move in the map according to its stored direction $c.dir$;

2. every robot c either updates its direction with the one stored in the node $@(c)$ if the latter is more recent (i.e., if $@(c)\,.ts > c.ts$) or overwrites the direction stored in the node with its own otherwise.

4.3.3 Reconfigurable ring

Consider a system of nodes arranged in a ring topology where a passing token allows each node, in turns, to communicate with the next. This rather simple coordination scheme is enriched with two dynamic elements characterizing the system:

1. new nodes are created and added to the ring constantly, until it reaches a given size limit N;

2. nodes can fail and get removed from the ring at a rate that increases with the ring size.

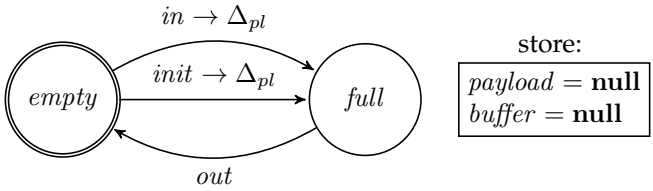


Figure 22: The $Node$ component type

The $Node$ component type is represented in figure 22. Each $Node$ has two local variables: $payload$, that holds the next value to send, and $buffer$, that stores the last value received. The $payload$ variable is initialized via the operation Δ_{pl} associated to the transitions that change the control location of a $Node$ from $empty$ to $full$, which will contain an assignment of the form $payload := v$, where v is a value that the $Node$ wants to transfer to its neighbor. The out and in ports are used intuitively to model the send and receive actions that a $Node$ can perform with its neighbors in the ring, while the $init$ port is used to handle more conveniently the bootstrapping of the system when all nodes are in the initial $empty$ control location.

The motif that will model the system needs to be equipped with an appropriate map to represent the ring topology of interest, which is essentially a cyclic directed graph with just one cycle. The functions associated to this kind of map will have to allow adding and removing map locations by also updating the set of edges in order to preserve the initial properties of the graph. Furthermore, we assume that initially each $Node$ instance has a one-to-one correspondence with a location. Figure 23 illustrates a graphical representation of the ring with four $Node$ instances, where the data flow and relationship between component instances and locations of the map are highlighted in green. Having these ingredients, we can define a coordination term r that realizes the described behavior adopting the conjunctive style. This allows us to be compositional in our design process, so we will make use of this advantage and divide the problem of defining r into three simpler sub-problems.

First, we will model how the ring grows and shrinks as new $Node$ instances are created and removed. As we previously mentioned, we

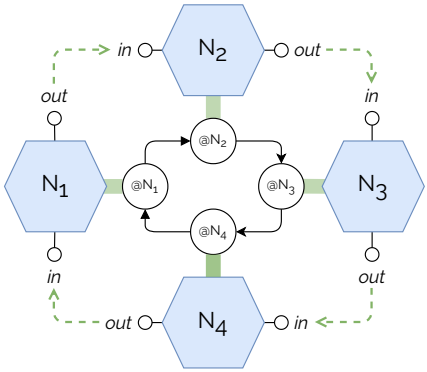


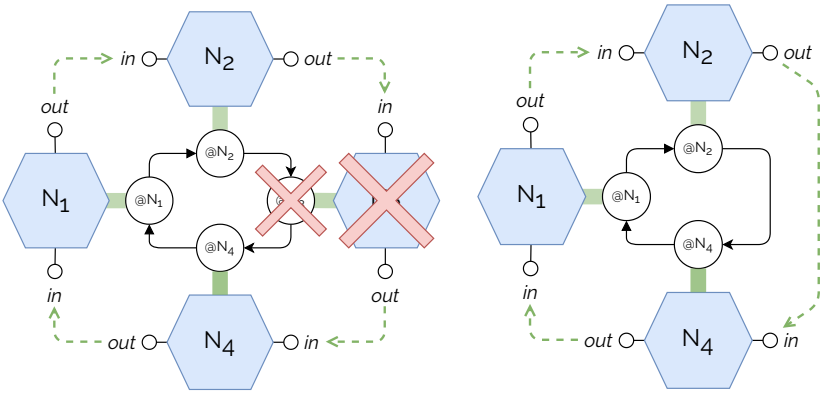
Figure 23: A representation of a ring with four *Node* instances and the corresponding map

will design the system in such a way that one new *Node* is added to the *ring* map as long as its size is less than N. At the same time, any *Node* instance in the system can terminate and get removed from the ring with probability $D\left(\frac{ring.size}{N}\right)^2$, where $D \in [0,1]$. When this happens, the ring will automatically reconfigure to preserve the overall structure as illustrated in figure 24. To keep the example simple, we encode the information required to model this behavior directly within the motif: the constants N and D will be statically defined within the coordination term r, and the current ring size will be obtained from a property of the map (i.e., the number of its locations). We can encode this behavior with the following rule:

$$
\begin{aligned}
r_1 = {} & ring.size < N \rhd \mathbf{true} \\
& \quad \rightarrow \left\{ \mathsf{addNode}(n, \mathtt{this}, g) \left[\mathsf{create}(Node, \mathtt{this}, n) \right] \right\} \\
\& \\
& \forall c : Node \left\{ rand < D\left(ring.size/N\right)^2 \rhd \mathbf{true} \right. \\
& \quad \left. \rightarrow \left\{ \mathsf{rmNode}(@(c), \mathtt{this}) \right\} \right\}
\end{aligned}
$$

where $rand \in [0,1]$ is a random number.

Next, we will define how components interact and transfer data. Communication is binary between neighboring nodes in the ring: a *Node* n_i



(a) N_3 and the location $@(N_3)$ are deleted

(b) and the map reconfigures to preserve the *ring*

Figure 24: How the system handles *Node* deletion

can send data to a *Node* n_j only if the edge $(@(n_i), @(n_j))$ belongs to the *ring* map (which we express with the predicate $\mathsf{isEdge}(n_i, n_j)$). Additionally, the node sending data and the one receiving it have to participate in the interaction with ports *out* and *in*, respectively. We can encode these constraints in the following way:

$$
\begin{aligned}
r_2 = {} & \forall c_1, c_2 : \mathit{Node} \{ c_1.\mathit{out} \rhd \mathsf{isEdge}(c_1, c_2) \wedge c_2.\mathit{in} \rightarrow \mathbf{skip} \} \\
& \& \\
& \forall c_1, c_2 : \mathit{Node} \{ c_1.\mathit{in} \rhd \mathsf{isEdge}(c_2, c_1) \wedge c_2.\mathit{out} \\
& \qquad \rightarrow \{ c_1.\mathit{buffer} := c_2.\mathit{payload} \} \}
\end{aligned}
$$

Lastly, we need to handle the transient situations when all nodes are in the *empty* (initial) control location. We can model a system with a single passing token by defining a coordination term that allows just one *Node* instance to perform an *init* provided that every other *Node* instance is *empty* and stays *idle*:

$$r_3 = \forall c_1, c_2 : \mathit{Node} \{ c_1.\mathit{init} \rhd c_1 = c_2 \vee (c_2.\mathit{empty} \wedge c_2.\mathit{idle}) \rightarrow \mathbf{skip} \}$$

This term will come into play after the creation of the very first *Node*

instance in the ring and in the event that the $Node$ instance holding the token gets removed from the ring.

The overall coordination rule r of the motif modeling the system will be the conjunction of the given rules:

$$r = r_1 \,\&\, r_2 \,\&\, r_3$$

4.3.4 Simple platooning protocol for automated highways

Platooning protocols aim at addressing the problem of coordinating collections of vehicles autonomously while guaranteeing some desired properties (e.g., human safety, fuel efficiency, traffic intensity, and so on). Here we will model a simple platooning protocol inspired by [10, 30].

Let us consider a scenario where we have a one-way road where autonomous cars move. Cars are organized in *platoons*, i.e., groups of one or more cars next to one-another proceeding at the same speed. Each platoon has one *leader* car, that is the car leading the platoon. While moving, a platoon can:

- *join* another platoon, if the distance between their leading/trailing cars is below a certain threshold d: after the platoons join, the cruising speed is set to the value of the slower one (i.e., the one that was on the front);
- *split* into two separate platoons, if for instance a car has to decrease its speed in order to exit the road via an intersection: the speed of the platoon in front is increased by a factor Δ_s, while the one in the back is decreased by the same factor Δ_s.

To model the system we define two motif types: the $Road$ and the $Platoon$. The $Platoon$ motifs define how cars bound to the specific motif instance move and keep track of their relative position within the platoon. One $Road$ motif instance represents the whole system, and coordinates all the $Platoon$ instances it hosts.

We model cars as component instances of the type Car. A car keeps track of its *speed* and *position* on the road, and has an interface towards

other cars to signal the initiation of a joining/splitting procedure, the acknowledgment of another car's joining/splitting, and the completion of either procedures. Figure 25 provides a complete representation of the Car component type.

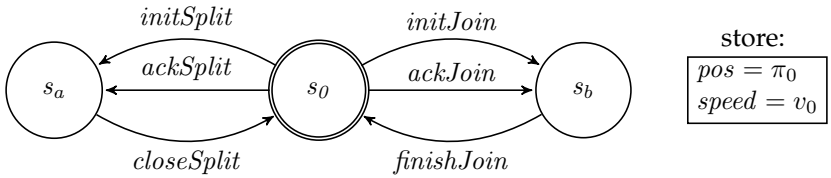


Figure 25: The Car component type

The $Platoon$ motif type is characterized by a map that models the train of cars using an indexed linked list, where each location is referenced by a unique integer address and is meant to be occupied by one car (figure 26).

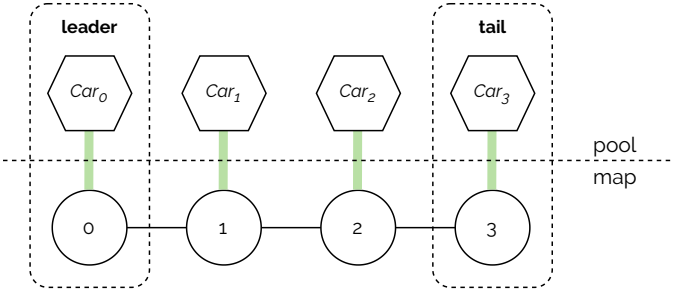


Figure 26: A representation of the $Platoon$ map

The coordination rule r_P associated to the motif in this simple scenario defines:

1. how each car in the platoon updates its position based on its current speed;

2. the requirement of having more than 1 car for any of them to trigger the splitting of the platoon;

3. the synchronization between all cars in the platoon when the joining/splitting maneuvers are performed.

This can be encoded in the DReAM coordination language using the conjunctive style as follows:

$$r_P = \forall c : Car\big\{\mathbf{true} \to \{c.pos := c.pos + c.speed \times \Delta_t\}\big\} \tag{4.26}$$

$$\&$$

$$\forall c : Car\big\{c.initSplit \rhd \mathsf{poolSize}(\texttt{this}) > 1 \wedge \mathsf{leader}(\texttt{this}) \neq c \wedge \mathsf{rand}(0,1) < p_{split} \to \mathbf{skip}\big\} \tag{4.27}$$

$$\&$$

$$\forall c : Car\big\{\forall c' : Car\big\{c.ackJoin \rhd c'.ackJoin \to \mathbf{skip}\big\}\big\} \tag{4.28}$$

$$\&$$

$$\forall c : Car\big\{\forall c' : Car\big\{c.initJoin \rhd c'.initJoin \to \mathbf{skip}\big\}\big\} \tag{4.29}$$

$$\&$$

$$\forall c : Car\big\{\forall c' : Car\big\{c.finishJoin \rhd c'.finishJoin \to \mathbf{skip}\big\}\big\} \tag{4.30}$$

$$\&$$

$$\forall c : Car\big\{\exists c' : Car\big\{c.ackSplit \rhd c'.initSplit \to \mathbf{skip}\big\}\big\} \tag{4.31}$$

$$\&$$

$$\forall c : Car\big\{\forall c' : Car\big\{c.initSplit \rhd c'.ackSplit \vee c = c' \to \mathbf{skip}\big\}\big\} \tag{4.32}$$

where:

- $\mathsf{poolSize}(m)$ is a function that returns the number of components and motifs hosted in m;
- $\mathsf{leader}(m) \in \mathcal{F}$ is a function of the $Platoon$ map that returns the entity assigned to the first position (0-indexed) of the map of motif m;
- $\mathsf{rand}(0,1)$ is a function that returns a random number between 0 and 1;

– p_{split} is the probability that a car can initiate a splitting maneuver at any given time (for simplicity we are assuming this being a parameter of the system).

Sub-rules (4.26-4.27) directly implement the policies described in 1-2. Sub-rules (4.28-4.30) force the symmetric synchronization of all the cars in the platoon when a join maneuver is initiated, acknowledged, or completed. Lastly (4.31-4.32) prescribe a splitting platoon to have all its cars but one acknowledge the split that the latter is initiating.

As we are keeping track of the positions of cars with local variables, the topological features of the *Road* motif map are not used. This simplifies the rules of the motif and saves us from having to apply discretization to the space of positions.

We start by defining the rules that realize the *join* procedure. The general idea is that when a platoon approaches another one ahead of it, the former adjust its speed and then every car in it migrates to the latter. This procedure is thus broken into two parts:

1. the first part defines how cars of a platoon initiate a join procedure, which requires that the leading car of the platoon is "close enough" to the tailing car of another platoon (i.e., the distance must be below a given threshold Δ_j) and that its leader acknowledges the join:

$$
\begin{aligned}
r_j^1 = \forall p : Platoon\big\{ & \forall c : p.Car\big\{ \exists p' : Platoon\big\{ \\
& c.initJoin \rhd (0 < \mathsf{tail}(p').pos - \mathsf{leader}(p).pos \leq \Delta_j) \\
& \wedge\, p \neq p' \wedge \mathsf{leader}(p').ackJoin \\
& \rightarrow \big\{ \mathsf{addNode}(p', c)\,, \\
& \quad c.speed := \mathsf{leader}(p').speed, \\
& \quad @(c).newLeader := \mathsf{leader}(p')\,, \\
& \quad @(c).newLoc := \mathsf{mapSize}(p') + @(c) \,\big\}\big\}\big\} \\
\&\quad & \\
\forall p : Platoon\big\{ & \exists p' : Platoon\big\{ \\
& \mathsf{leader}(p)\,.ackJoin \rhd (0 < \mathsf{tail}(p).pos - \mathsf{leader}(p').pos \leq \Delta_j) \\
& \wedge\, \mathsf{leader}(p')\,.initJoin \rightarrow \mathbf{skip}\big\}\big\}
\end{aligned}
$$

Notice that when the conjunctive term of the first subformula is satisfied, the following operations are performed:

- a new location is created in the map of p';
- the variable $c.speed$ of the car in the rear platoon p is set to the speed of the leading car in the platoon on front (p');
- the map memory at location $@(c)$ is used to store temporary information regarding the identity of the leader car in the platoon on the front and the new location index that the car will occupy in the map of p' (note here that $@(c)$ in the sum operation is overloaded and used here as the index of the location where c is mapped in platoon p).

2. the second part defines how the join is finalized with the involved cars migrating to the platoon on the front:

$$
\begin{aligned}
r_j^2 = \forall p : Platoon \{ \forall c : p.Car \{ \exists p' : Platoon \{ \\
& c.finishJoin \rhd \mathsf{leader}(p').finishJoin \\
& \wedge (\mathsf{leader}(p') = @(c).newLeader \vee p = p') \\
& \rightarrow \{ \mathsf{IF}\ (p \neq p')\ \mathsf{THEN}\ \{ \\
& \quad \mathsf{migrate}(c, p', @(c).newLoc), \mathsf{delete}(p) \} \} \} \} \}
\end{aligned}
$$

The requirement of the conjunctive term guarantees that the join is finalized with the appropriate platoon, and the operations performed migrate each car to the stored new location before deleting motif p (notice that the delete operation will be carried out once for each car in motif p, but the effect will be equivalent as if only one delete operation is performed).

The platoon *splitting* procedure is more complex as it requires the creation of one new $Platoon$ instance with the appropriate number of locations to host the cars leaving their original platoon before any migration between the two can effectively happen. We propose a possible way to model this procedure by breaking it into three conjunctive rules:

1. the first rule defines the capability of any car in a platoon to initiate the splitting maneuver, resulting in the creation of a new platoon:

$$
\begin{aligned}
r_s^1 = \forall p : Platoon \big\{ \forall c : p.Car \big\{ & \\
& c.initSplit \rhd \mathbf{true} \\
& \rightarrow \big\{ c.speed := c.speed \times (1 - \Delta_s) \,, \\
& \quad \mathsf{create}(p' : Platoon, \mathtt{this}, @(p) \,, g) \, \big[\\
& \quad \mathsf{addNode}(n, p', c) \, \big[\mathsf{migrate}(c, p', n) \, \big] , \\
& \quad \mathsf{FOR} \, \big(i \in (@(c) \,, \mathsf{mapSize}(p)) \big) \, \mathsf{DO} \, \big\{ \mathsf{addNode}(p', i) \, \big\} \big] , \\
& \quad \mathsf{rmNode}(@(c) \,, \mathtt{this}) \, \big\} \big\} \big\}
\end{aligned}
$$

   When a car initiates a splitting maneuver, first it decreases its speed, then a new platoon p' is created. Within the context of p', a new location is created to which the car migrates, and then a "for" loop adds one node for each remaining nodes in the map of platoon p positioned after the one hosting the component that initiated the split procedure. Finally, outside the local scope of the operations on the new motif instance, the location where the car c was originally assigned is removed;

2. the second rule defines how the cars of a platoon involved in a splitting maneuver synchronize and acknowledge its initiation:

$$
\begin{aligned}
r_s^2 = \forall p : Platoon \big\{ \forall c : p.Car \big\{ \exists c' : p.Car \big\{ & \\
& c.ackSplit \rhd c'.initSplit \\
& \rightarrow \big\{ \mathsf{IF} \, \big(@(c) > @(c') \big) \, \mathsf{THEN} \, \big\{ \\
& \quad @(c).newLeader := c, \\
& \quad @(c).newLoc := @(c) - @(c') \,, \\
& \quad c.speed := c.speed \times (1 - \Delta_s) \\
& \big\} \, \mathsf{ELSE} \, \big\{ c.speed := c.speed \times (1 + \Delta_s) \, \big\} \big\} \big\} \big\}
\end{aligned}
$$

   Every car acknowledging a split needs to account for it in a different way depending on its position in the platoon relative to the car that

initiates the maneuver. To implement this, an "if" conditional statement in r_s^2 operations allows only cars in the back of the splitting point to store the new leader and location of the rest of the splitting cars, while also modifying their speed by a factor of $(1 - \Delta_s)$. Collaterally, the rest of the cars not leaving the platoon increase their speed by a factor of $(1 + \Delta_s)$;

3. the last rule completes the procedure by migrating all the cars that were behind the one which initiated the splitting to the platoon where the latter is now leading:

$$
\begin{aligned}
r_s^3 = \forall p : Platoon \big\{ \forall c : p.Car \big\{ \exists p' : Platoon \big\{ \\
& c.closeSplit \rhd @(c).newLeader = \mathbf{null} \\
& \vee (\mathsf{leader}(p').closeSplit \wedge @(c).newLeader = \mathsf{leader}(p')) \\
& \rightarrow \big\{ \mathsf{IF}\ \big(@(c).newLeader \neq \mathbf{null} \big)\ \mathsf{THEN}\ \big\{ \\
& \qquad \mathsf{migrate}(c, p', @(c).newLoc)\,, \\
& \qquad \mathsf{rmNode}(@(c)\,, p)\ \big\} \big\} \big\} \big\}
\end{aligned}
$$

The rule allows all splitting cars (including the one that initiated the procedure) to synchronize via port $closeSplit$. Cars that are not already in the new platoon (i.e. those that have stored in their map node the information regarding their new leader and position) are finally migrated to their new location and their old location is removed.

Overall, the $Road$ motif will be characterized by a coordination rule obtained from the conjunction of the rules that we defined so far:

$$r_R = r_j^1 \ \&\ r_j^2 \ \&\ r_s^1 \ \&\ r_s^2 \ \&\ r_s^3$$

Chapter 5

Executable implementation

The DReAM framework has been implemented in Java as an execution engine with an associated library of classes and methods that support system specification conforming to the DReAM syntax. The first version of this implementation has been presented in [18].

This Chapter presents jDReAM, an all-new implementation of the framework rebuilt from the ground up in order to support both DReAM and L-DReAM specifications. Currently jDReAM consists of roughly 200 classes organized in more than 20 packages. A detailed documentation of all its constituent elements is beyond the scope of this Thesis: here we discuss only the main building blocks of the platform. We first illustrate the "core" architecture of the solution which implements the execution engine and defines the foundational classes required to encode L-DReAM specifications. We then describe an additional set of packages and libraries that extend the core module in order to support DReAM specifications. While the development of jDReAM is still active, the source code is available on GitHub and accessible at the following url: `https://github.com/ZephonSoul/j-dream`.

To represent the main elements of jDReAM and their relationships we will adopt a graphical syntax coherent with UML conceptual class diagrams (see figure 27). To ensure a readable presentation of each key concept, each diagram is going to focus on a specific "package" of the

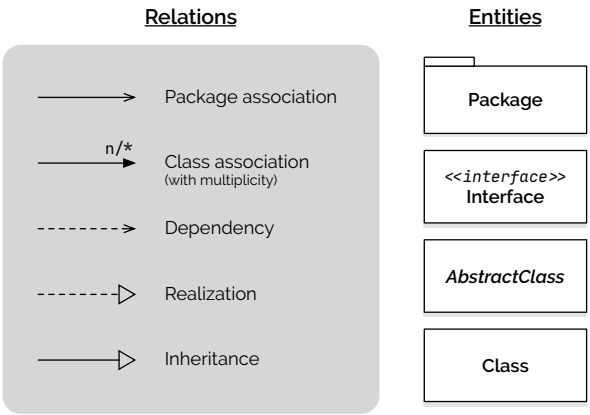


Figure 27: Reference schematics syntax inspired by UML class diagrams

jDReAM library. Classes and interfaces that are detailed in specific diagrams are color-coded across the different figures where they are referenced.

To show the effectiveness of our framework, we use jDReAM to implement, execute and evaluate relevant example systems discussed in Chapters 4. The classes implementing the simple use cases described at the end of this Chapter are also available in the project repository.

5.1 The jDReAM core architecture

The key elements that constitute the "core" of the jDReAM implementation are illustrated in figure 28, and comprise:

- the `ExecutionEngine` class, which implements the operational semantics;
- the `execution strategies` package, which leverages the Strategy design pattern [23] to implement different approaches for admissible interaction selection;
- the `output handlers` package, providing helper classes for the `ExecutionEngine` logging and presentation facilities;

– the `entities` package, containing all the classes that define the structure of L-DReAM components;

– the `coordination` package, which provides the implementation of all the elements of the coordination language as Java artifacts, including PIL formulas and operations;

– the `expressions` package, that mainly supports the classes of the `coordination` package by implementing arithmetic and set-theory expressions.

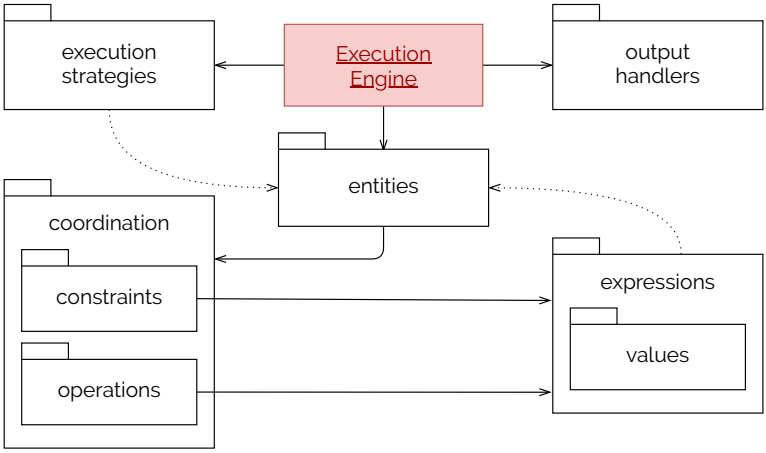


Figure 28: Package overview of the jDReAM core architecture

This set of software packages supports the specification of executable systems conforming to the L-DReAM abstract syntax which evolve according to its operational semantics described in Chapter 3.

To have a more precise idea of the inner workings of jDReAM, we will illustrate the contents and function of the two main packages that constitute the core architecture: the `entities` and `coordination` packages.

The `entities` package

Being an object-oriented programming language, the class-object dualism in Java is conceptually very close to the type-instance pair of L-DReAM components. This is also one of the reasons that guided the choice of this programming language as a first platform to develop jDReAM instead of selecting other options using different paradigms that would have been perhaps a better fit for the implementation of the underlying coordination logic. To leverage Java's native mechanism of generalization, each

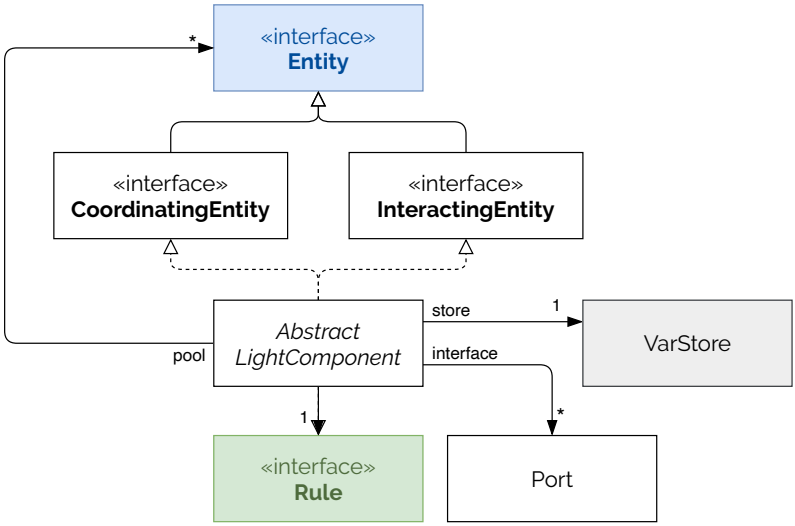


Figure 29: Simplified class diagram of the `entities` package

L-DReAM component type corresponds to a Java class. To streamline their definition, each user-specified component type must implement an `Entity` *interface* that encapsulates all the required methods for a component instance to work with jDReAM. Two extensions to the `Entity` interface are provided:

1. `CoordinatingEntity`: implemented by compound entities (i.e., objects that host a pool of other entities and have coordination rules to define how they interact);

2. `InteractingEntity`: implemented by entities that are equipped with an interface to interact with other entities.

Many of these functions, common to all L-DReAM components, are factorized in an *abstract class* `AbstractLightComponent` which implements both interfaces. Additionally, it also provides a number of methods contributing to the implementation of the semantics of L-DReAM. This allows the user to extend the abstract class and just override its constructor by leveraging the ones in its superclass.

The interface and pool of a L-DReAM component are implemented as Java collections of `Port` and `Entity` instances, while the store is realized with a dedicated `VarStore` Java class composed by `LocalVariable` instances. All of them are defined within the `AbstractLightComponent` class as *instance variables*.

Lastly, the L-DReAM rule that defines the component behavior and the coordination constraints over the components in its pool is also stored in an instance variable of `AbstractLightComponent` and it is described via the `Rule` interface of the `coordination` package.

Figure 29 presents the main elements of the `entities` package that have just been described in the form of a simplified class diagram.

The `coordination` package

The `coordination` package contains a collection of classes and subpackages that implement the L-DReAM language itself. At top-level it provides the `Rule` interface, which represents any L-DReAM rule that can be built using the syntax in (3.33). Concretely, a `Rule` can be either instanced as a `Term` (either disjunctive or conjunctive), a `FOILRule` (i.e., a `Rule` bound to a `Declaration`), or a combination of rules in the form of either a conjunction (`AndRule`) or disjunction (`OrRule`). This is illustrated in figure 30.

Classes implementing terms contain instance variables that define constraints on the interaction and local variables, as well as operations that are performed when the former hold. These two concepts are abstracted through the use of the `Formula` and `Operation` interfaces, respectively.

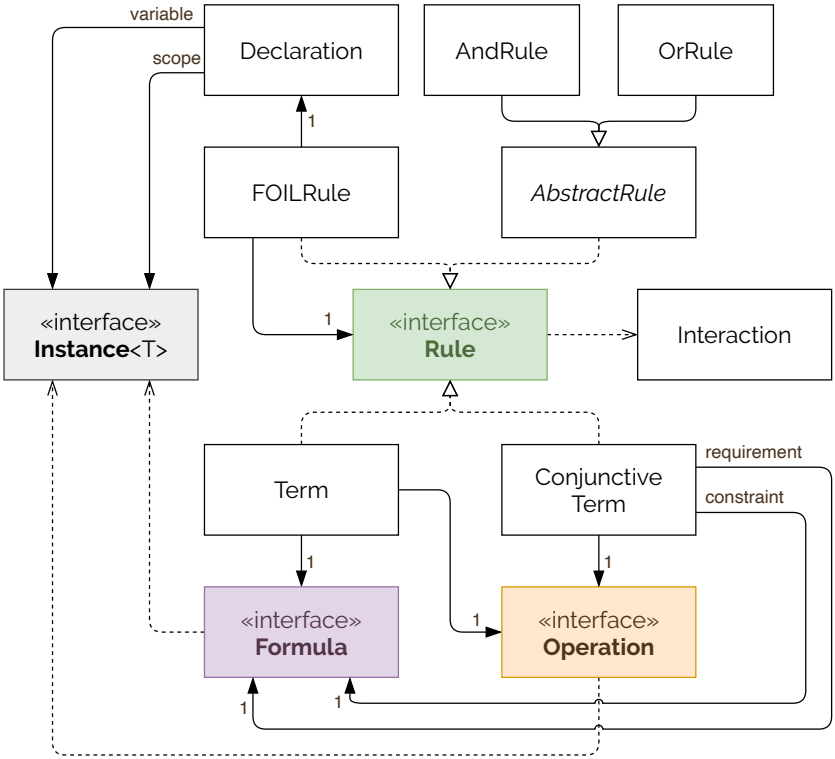


Figure 30: Simplified class diagram of the `coordination` package

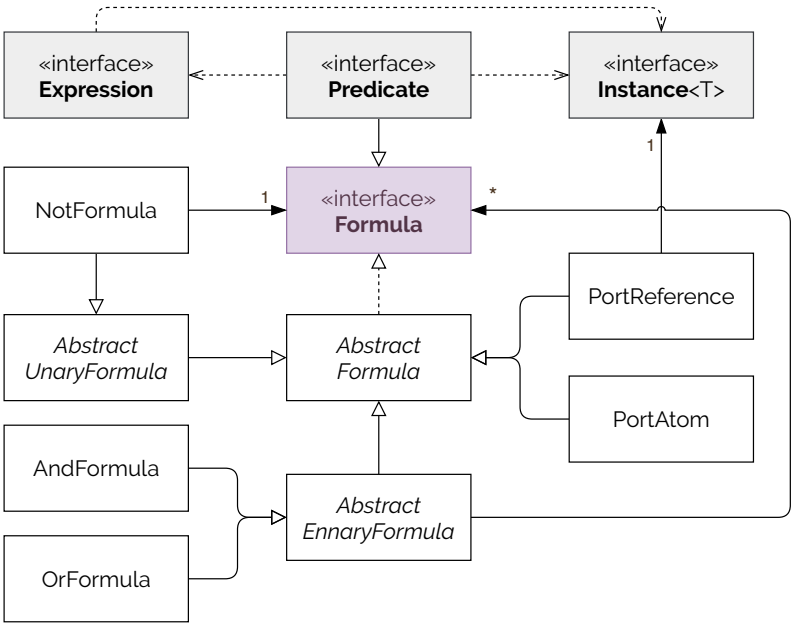


Figure 31: Simplified class diagram of the `constraints` package

Classes that implement these interfaces are dependent on another interface: the `Instance<T>`. This parametric interface decouples references to individual elements of the framework from their actual Class instance. In the scope of L-DReAM, this is used to define component variables and local variable references when defining the coordination rules.

Figures 31 and 32 expand the contents of the `constraints` and `operations` sub-packages presenting the main classes that implement the `Formula` and `Operation` interfaces. It is worth noting that many elements of these packages have a dependency towards the `Expression` interface, which encapsulates the facilities offered by the `expressions` package providing the classes that implement algebraic expressions over local variables and values in L-DReAM.

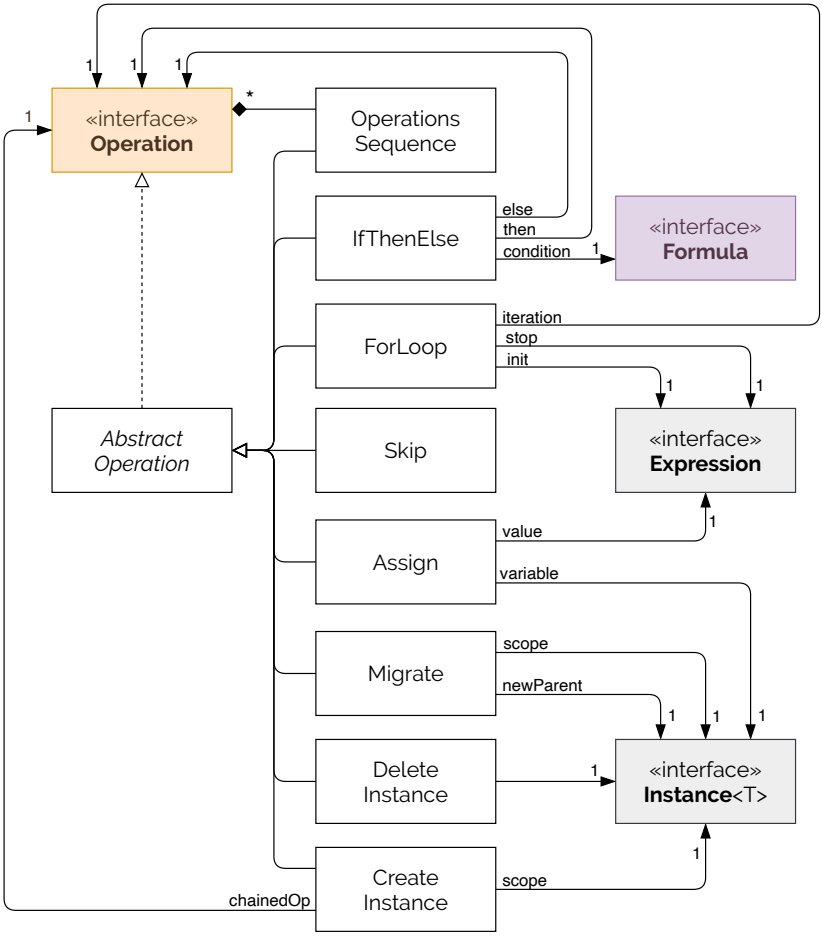


Figure 32: Simplified class diagram of the `operations` package

5.2 The jDReAM extended architecture

Since DReAM can be considered a specialized version of L-DReAM, the core libraries of jDReAM could already be used as-is to implement DReAM specifications with very little approximation. This would however prove to be inefficient, as many choices that have been made in order to tailor DReAM to more "concrete" systems also allow many opportunities for optimization and complexity reduction. One of such examples is the choice of defining the behavior of atomic components with a labeled transition system instead of relying uniquely on coordination rules, which add to the complexity of the global coordination model that the execution engine has to build and solve at each execution cycle.

To take advantage of these opportunities and have an API for DReAM that is closer to its Java implementation, the core packages presented in section 5.1 are extended to encompass DReAM's structuring concepts such as motifs and maps.

Given the number of shared concepts between the two theoretical frameworks and the modularity of the elements of the core architecture, many of them translate directly to the extended architecture and are simply "imported".

There are however substantial additions to three packages presented in figure 28: the `entities`, `operations` and `expressions` packages. In the following diagrams new/modified classes are marked with a red bookmark to distinguish them from the ones belonging to the jDReAM core.

The extended `entities` package

In L-DReAM, components seamlessly act as both coordinating and interacting entities depending on their definition and state. This is reflected by the implementation of the `AbstractLightComponent` class which extends both `CoordinatingEntity` and `InteractingEntity` interfaces in jDReAM (see figure 29). In DReAM, motifs and components implement one of the two interfaces each. Two abstract classes (`AbstractMotif` and `AbstractComponent`) implement com-

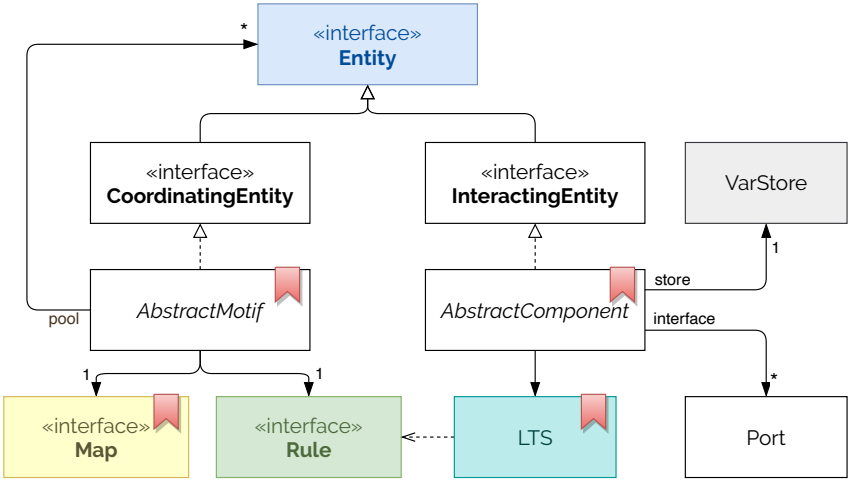


Figure 33: Simplified class diagram of the extended `entities` package

mon methods and instance variables relevant to the respective concepts of the framework. Notably, `AbstractComponent` now relies on an `LTS` object to define its behavior and interaction capabilities instead of using a general coordination rule, and `AbstractMotif` maintains a reference to an associated `Map` object in an instance variable. These extensions are summarized in figure 33 (where we omitted classes and relationships only relevant to L-DReAM). The `LTS` class (figure 34) is composed of `Transi-`

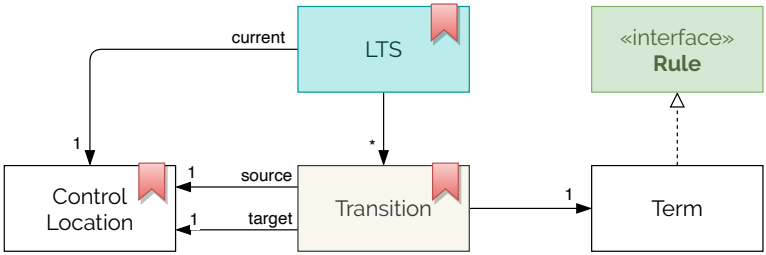


Figure 34: Class diagram of the `LTS` class

`tion` objects, which directly model DReAM transitions associating two

`ControlLocation` objects (i.e., source and target of the transition) with a t-PILOps rule, which is implemented with a static `Term` of the coordination language.

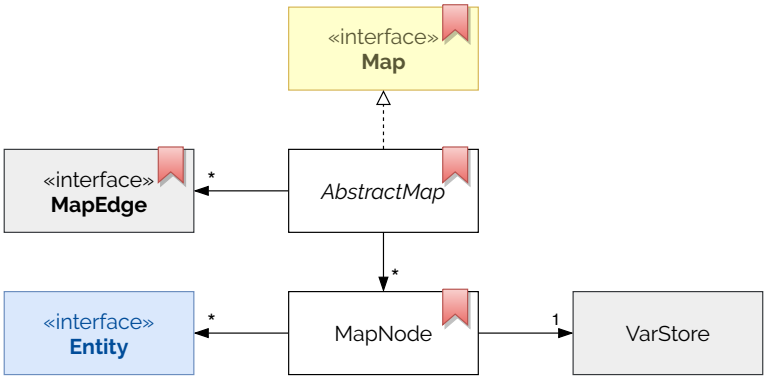


Figure 35: Class diagram of the `maps` package

At a general level, DReAM maps are graphs with memory associated to each node, but in practice we have shown how having specific map implementations enriched with utility functions allows to simplify the definition of coordination rules and system modeling. The `maps` package (figure 35) therefore defines a general interface that every "map" realization has to implement, and an `AbstractMap` that defines instance variables and utility methods shared among most `Map` implementations. `MapNode` instances that model map locations store references to objects implementing the `Entity` interface (i.e., either motifs or components associated to the location) and are equipped with a `VarStore` just like components.

The extended `operations` package

As both frameworks share the same underling coordination language, the `coordination` package and its `constraints` sub-package do not require substantial extensions to support DReAM specifications.

The `operations` package, on the other hand, is enriched with variants of the `Migrate` and `CreateInstance` operations that implement

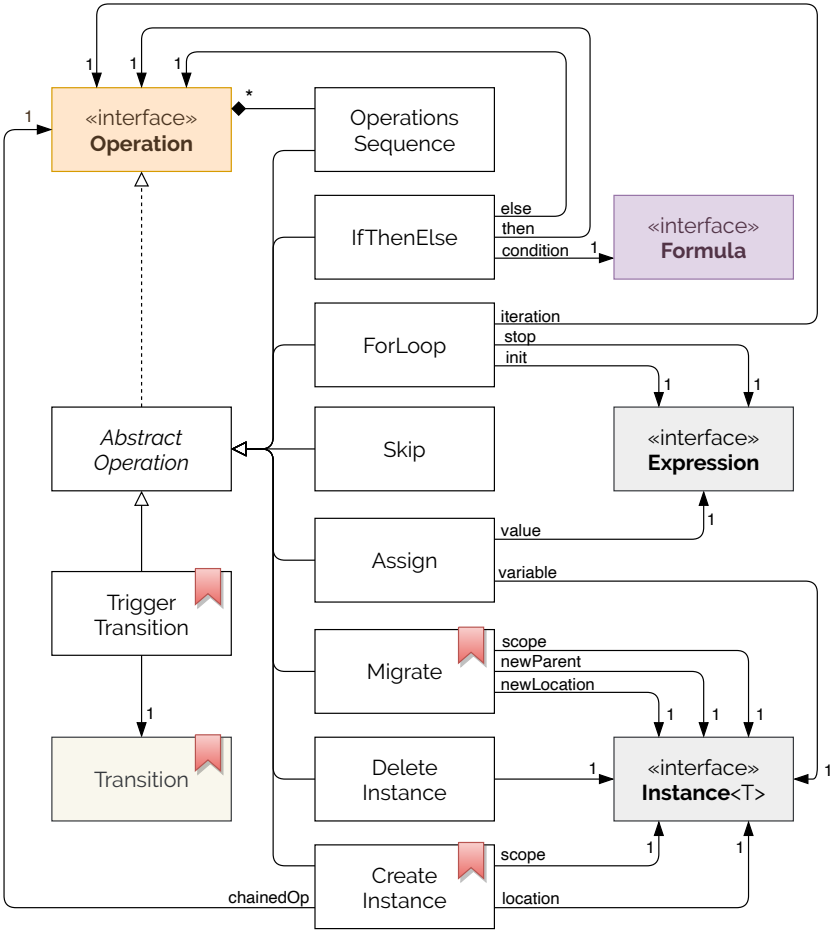


Figure 36: Simplified class diagram of the extended `operations` package

map reconfigurations (e.g., creation/deletion of nodes and edges), as represented in figure 36. Notice that references to `MapNode` objects are handled with classes extending the parametric `Instance<T>` interface like for entities and local variables. An additional `TriggerTransition` operation is also defined in order to handle firing of components' transitions uniformly with other operations.

The extended `expressions` package

As previously mentioned, the classes in the `expressions` package implement arithmetic and set-theory expressions. These involve constant values and sets of values, but also handle references to them in the form, for instance, of local variables.

In the core architecture of jDReAM, few classes depend on this package, mainly the ones implementing the `Predicate` interface and the `Assign` class realizing the assignment operation.

In the extended jDReAM architecture, the `expressions` package is enriched with additional classes in order to handle the new structuring concepts added at the `entities` level, i.e., maps and control locations in the behavior of components.

5.3 Use cases in practice

5.3.1 Coordinating flocks of robots

Recall the two examples presented in 4.3.1 and 4.3.2 describing two systems of robots exploring a bi-dimensional space trying to converge to a single "flock" where all its members move in the same direction.

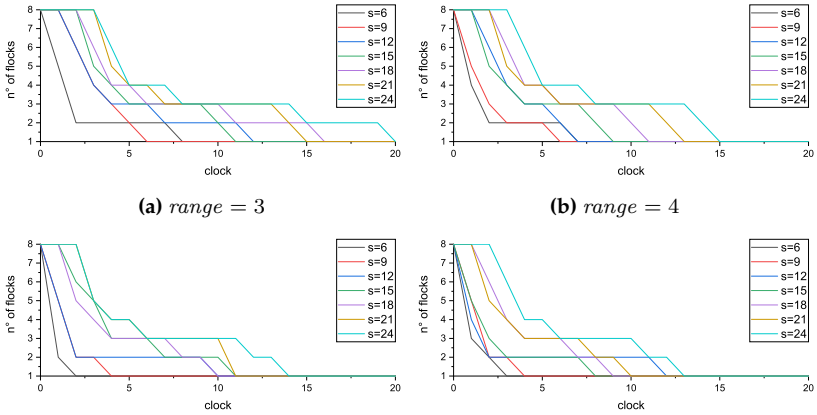
We used jDReAM to implement both variants (with and without stigmergy) and studied their behavior with different initial settings. For comparison purposes, in both cases we fixed the number of robots in the system to 9 and we chose a specific initial direction for each one of them. The mapping of the robots to a grid of size $s \times s$ is realized in such a way that they are uniformly spaced both horizontally and vertically. We chose grid sizes proportional to 3 for uniformity.

Refer to section B.1 of the Appendix for the listings of the Java classes defined using jDReAM for the "interacting robots" example.

Interacting robots without stigmergy

In this first case, we executed the system multiple times while varying the size s of the grid and the communication range for a fixed number of robots. During each execution, we monitored the number of flocks (i.e., the number of groups formed by robots moving in the same direction). Intuitively, we expect to observe a faster convergence in the movement directions as the size of the grid shrinks and/or as the communication range increases.

The graphs in figure 37 show the trend in the number of flocks over time for different system setups. Indeed, the results confirm our expec-



(a) $range = 3$ **(b)** $range = 4$

(c) $range = 5$ **(d)** $range = 6$

Figure 37: Trends in the number of flocks over time at different communication ranges

tations: the adopted initial setup procedure of the robot's positions and directions allows them to converge to an homogeneous flock within 20 clock ticks, a number which decreases as we increase the communication range. There is also an opposite trend when increasing the size of the

grid, although it is interesting to see that there are several exceptions to this rule (e.g. for $range = 3$ convergence on the grid $s = 6$ takes more time than on the grid $s = 9$; the same applies for $s = 12$ vs $s = 15$ and $s = 18$ vs $s = 21$).

Coordinating robots with stigmergy

For a comparison with the systems of 5.3.1, we fixed the same parameters regarding number of robots, initial directions, set of tested grid sizes and mapping criterion also in the case of robots coordinating using stigmergy.

We expected a similar correlation between convergence time and grid size as in the case for communicating robots. Indeed, this is confirmed by the graph in figure 38, which shows the trends in the number of flocks for different grid sizes.

It is worth observing that convergence time and grid size are, again, not always directly proportional: here it is noticeable how the robots converge to a single flock for grid sizes equal to 15 and 21 in roughly half the time it takes for them to converge on the smaller grid with $s = 12$. The graphs in figure 39 compare directly the convergence trends

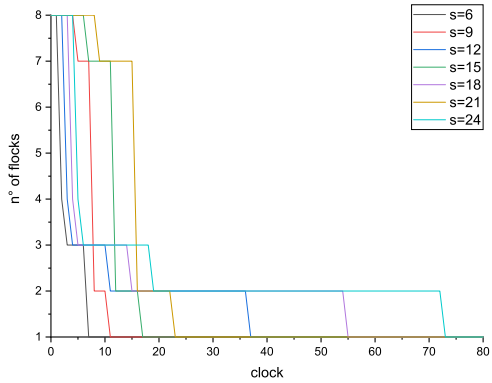
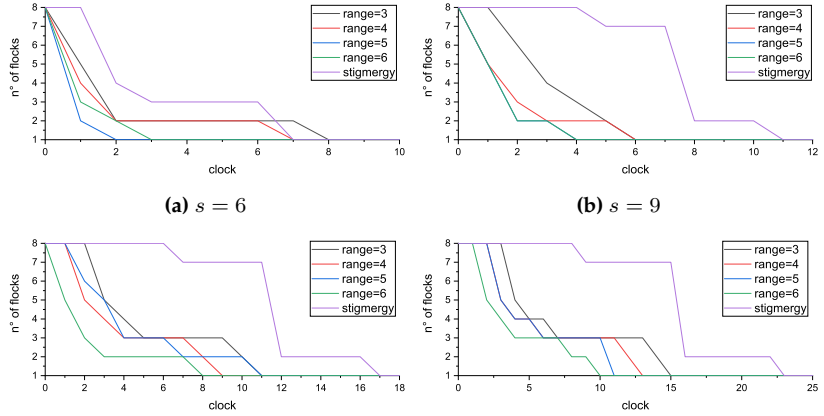


Figure 38: Evolution of the number of flocks over time at different communication ranges

using the two approaches on grids of different sizes. From these we

can appreciate how the stigmergy-based solution performs roughly on-par with the interaction-based one for small maps, progressively losing ground to the latter as the map becomes larger. This comparison also helps to better visualize how the implementation not resorting on sensors initially requires some time to populate the map with information which is proportional with the size of the map itself.



(a) $s = 6$ **(b)** $s = 9$

(c) $s = 15$ **(d)** $s = 21$

Figure 39: Comparison between the two approaches at different grid sizes

5.3.2 Reconfigurable ring

We executed the reconfigurable ring example presented in 4.3.3 using the jDREAM Java API. Like in the use cases modeling the flocks of robots studied in 5.3.1, we monitored a simple metric of the system to evaluate its overall behavior, but in this case we chose the size of the ring (i.e., the number of active *Node* instances). Choosing the system parameters $N = 20$ and $D = 0.5$, the probability that the ring will grow in size at each iteration can be easily computed as $p_g(n) = (1 - p_d(n))^n$, where n is the current size of the ring and $p_d(n) = 0.5 \cdot (n/20)^2$ is the probability that at least one node will get deleted. Table 1 displays the values of p_g for n ranging from 1 to N. Figure 40 visualizes the results produced by the

poolSize($ring$)	*Growth prob.*	poolSize($ring$)	*Growth prob.*
1	99.8750%	11	16.4656%
2	99.0025%	12	9.2420%
3	96.6628%	13	4.5731%
4	92.2368%	14	1.9555%
5	85.3215%	15	0.7058%
6	75.8613%	16	0.2090%
7	64.2465%	17	0.0490%
8	51.3219%	18	0.0087%
9	38.2605%	19	0.0011%
10	26.3076%	20	0.0001%

Table 1: Growth probability at varying ring sizes for $N = 20$ and $D = 0.5$

first 200 iterations of the execution engine. The graph shows a behavior that is in line with what we were expecting from the system for the given N and D: the probability of the ring to steadily grow towards the size cap falls below 25% once the population of nodes surpasses 10, and the likelihood that the growth is going to be negated by at least one *Node* instance getting deleted exceeds 95% when the size of the ring reaches 13.

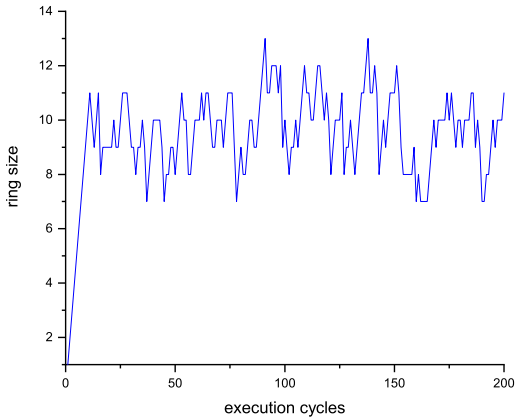


Figure 40: Evolution of the number of nodes in the ring

5.3.3 Simple platooning protocol

To evaluate the simple platooning protocol described in 4.3.4, we have constructed a set of test scenarios by choosing different values for the following initial parameters:

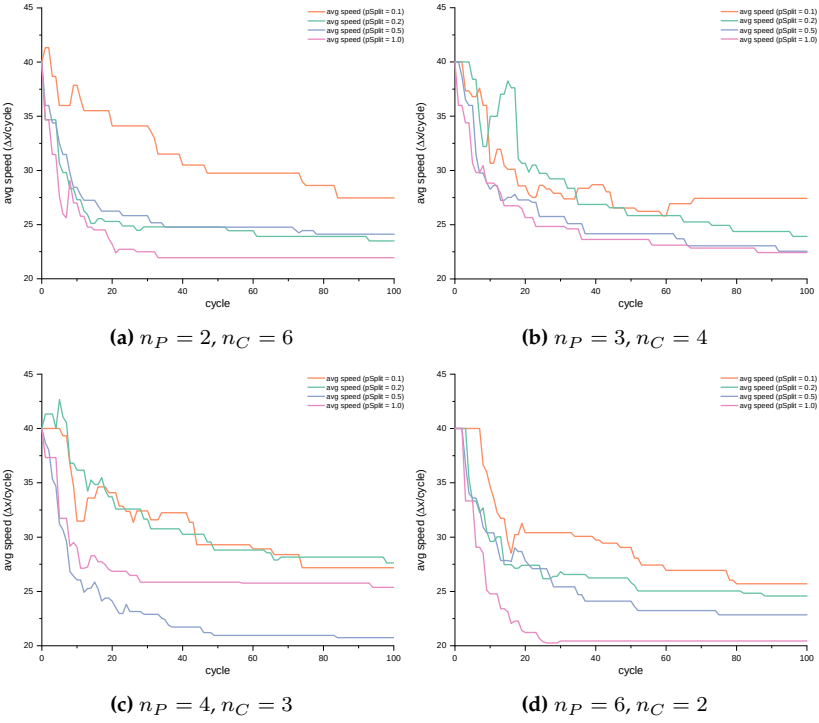
- n_P: the number of platoons in the system;
- n_C: the number of cars per platoon in the system (assuming n_C to be initially the same for all platoons);
- s_C: the speed of every car;
- d_C: the distance between two adjacent cars within a platoon;
- d_P: the distance between two adjacent platoons;
- Δ_j: the minimum distance between two adjacent platoons to trigger the "join" maneuver;
- Δ_s: the speed modification factor applied in the splitting maneuver;
- p_{split}: the probability that a car can initiate a "split" maneuver at any given time (provided it is not the leader of its platoon).

Additionally, the symmetry of the initial conditions can be altered by having s_C, d_C and d_P randomly distributed around their mean values with chosen variance.

We kept parameters n_P and n_C small enough to ease the analysis of the execution traces in relation to the behavior of individual components. From these traces we extracted, at each execution step, the number of distinct platoons on the road and both the position as well as the speed of each car. To this end, we opted to have a total of 12 Car instances initially split in $2, 3, 4$ and 6 different platoons.

We ran the system for 100 execution cycles with fully-symmetric initial conditions that guaranteed no triggering of joining maneuvers unless at least a platoon split happens first. This allows us to appreciate the effect of the split probability p_{split} on the behavior of the system.

Given how the coordination rules and behaviors are defined in 4.3.4, the only event that alters the cruising speed of cars in the system is the splitting maneuver. This is well represented in figure 41, showing the evolution of the average speed of cars in the system with different initial groupings into platoons. Although performing a split increases the speed



(a) $n_P = 2, n_C = 6$ **(b)** $n_P = 3, n_C = 4$

(c) $n_P = 4, n_C = 3$ **(d)** $n_P = 6, n_C = 2$

Figure 41: Average speed of cars in the system for different values of p_{split} ($s_C = 40$, $d_C = 15$, $d_P = 30$, $\Delta_j = 25$, $\Delta_s = 0.2$)

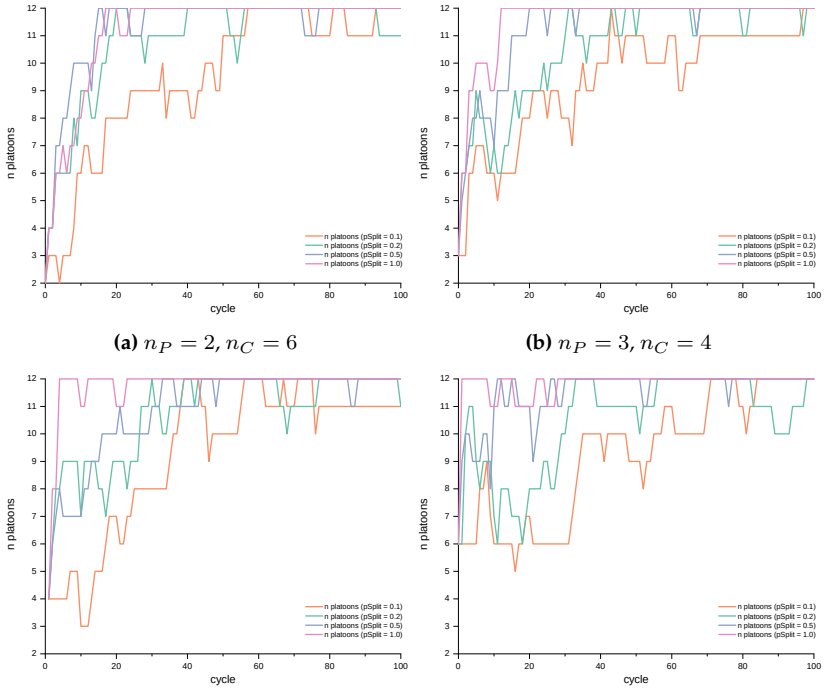
of the cars in front of the one initiating the maneuver, the cars slowing down are eventually reached by a trailing platoon. This causes more and more cars to slow down and join the slower platoon, and even if some cars can still split and increase their speed, the way we modeled this speed-up as a fixed ratio of the current speed makes regaining the lost velocity more difficult.

This behavior is confirmed across all the tested configurations of the system, where we observed an average speed decrease ranging from 30 to 50% after 100 execution cycles. As expected, lower split probabilities generally resulted in higher average final speed. It is interesting to note that the initial fragmentation of the same amount of cars in more distinct platoons does not change the relative speed trend dramatically, except for the corner case $p_{split} = 1$. Indeed, as leader cars cannot initiate a split maneuver, having more platoons effectively reduces the number of cars that can potentially trigger the procedure. E.g., for $n_P = 2$, 10 cars can potentially initiate a splitting, whereas for $n_P = 6$ this number reduces to 6. With two platoons of six cars each and an individual $p_{split} = 0.1$, the probability that at least one split happens is about 65%, whereas with six platoons of size two and double p_{split} the combined probability is less than 74% (where we computed the overall probability of at least one split happening for n eligible cars with $1 - (1 - p_{split})^n$).

Figure 42 shows the complementary picture of the evolution of the number of platoons in the system.

Another test of the protocol that can be easily performed is to fix the split probability and introduce noise in the initial configurations to simulate a more "realistic" scenario with cars entering the road at different speeds and not perfectly distanced between one-another. This was performed by sampling s_C, d_C and d_P from random uniform distributions $\mathcal{U}(a, b)$ (where a and b are the boundaries of the distribution). Figure 43 shows another way of visualizing the evolution of the resulting systems by plotting the individual positions of all cars at different initial settings for n_P and n_C.

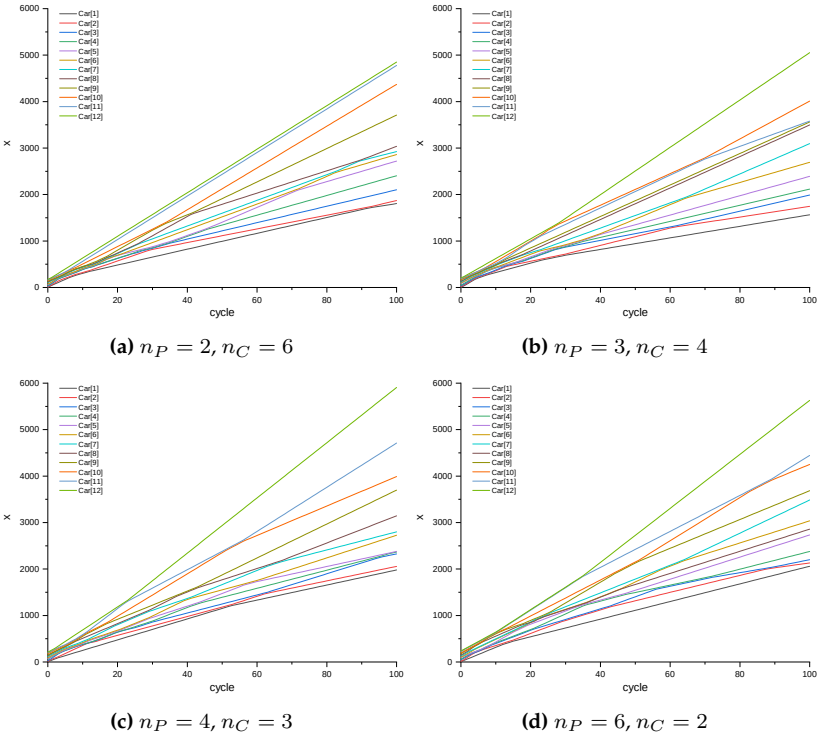
Refer to section B.2 of the Appendix for the listings of the Java classes defined using jDReAM to model the presented use case.



(a) $n_P = 2, n_C = 6$ **(b)** $n_P = 3, n_C = 4$

(c) $n_P = 4, n_C = 3$ **(d)** $n_P = 6, n_C = 2$

Figure 42: Number of distinct platoons in the system for different values of p_{split} ($s_C = 40$, $d_C = 15$, $d_P = 30$, $\Delta_j = 25$, $\Delta_s = 0.2$)



(a) $n_P = 2, n_C = 6$ **(b)** $n_P = 3, n_C = 4$

(c) $n_P = 4, n_C = 3$ **(d)** $n_P = 6, n_C = 2$

Figure 43: Positions of individual cars over 100 execution cycles ($s_C = \mathcal{U}(35, 45)$, $d_C = \mathcal{U}(12.5, 17.5)$, $d_P = \mathcal{U}(20, 40)$, $\Delta_j = 25$, $\Delta_s = 0.2$, $p_{split} = 0.2$)

Chapter 6

Concluding considerations

6.1 Summing up

The availability of adequate tools to develop complex systems that interact with one-another and have to react/adapt to unpredictable environments is a challenging feat, yet it is increasingly becoming a strict requirement as *ex post* correctness proves to be unmanageable to verify and major bugs get too expensive to fix. Addressing this issue at the architecture design level allows us to offer an adequate trade-off between the implementation-specific dependencies inherent to code programming and synthesis, and the general principles of "sound" design based on best practices and architectural patterns.

In this Thesis we have presented two frameworks for the description of dynamic reconfigurable systems supporting their incremental construction according to a hierarchy of structuring concepts. Both rely on the same coordination language inspired by propositional and first-order logic to define interactions and reconfigurations of components that constitute a system. We have shown how the expressive power of this formal language is sufficient to support many coordination patterns and to capture all the key features of dynamic systems. By not imposing *a priori* limits to the coordination and computation styles that can be represented with this language, we presented intuitions on how sound trans-

formations between different design approaches can be formalized. More specifically, we outlined methodologies to approach coordination of components using a compositional approach that we called *conjunctive style* - where the global constraint that defines admissible interactions is obtained from the conjunction of individual constraints associated to each component - and a connectors-centric approach that we called *disjunctive style* - where the whole interaction model is defined by the union of constraints that implement a specific coordination mechanism. At the same time, by studying the relationship between conjunctive and disjunctive styles, we have shown that while they are both equally expressive for interactions without data transfer, the disjunctive style is more expressive when interactions involve data transfer and reconfigurations.

L-DReAM provides the most "general" platform for architecture specification, as it captures the core elements of the underling modeling language with minimal overhead. DReAM is essentially a specialized version of L-DReAM that fixes some base design decisions and offers additional parametrization features, making the specification of dynamic and mobile architectures more straightforward and efficient.

Both frameworks model systems as aggregates of coordinating components. In L-DReAM these aggregates are completely symmetric: a component can host other components defining a hierarchy in which each aggregate (i.e., compound) defines the coordination rules that regulate the way in which other components downward in the hierarchy interact and evolve. DReAM systems are instead defined as hierarchies of motifs coordinating atomic components. Motifs represent independent dynamic architectures that define how their underlying components interact and reconfigure.

Both frameworks guarantee enhanced expressiveness and incremental modifiability thanks to the following features:

Incremental modifiability: Coordination rules associated with components in an aggregate (either a compound or a motif) can be modified and composed independently. Components can be defined independently of their context of use. Self-organization can be modeled by combining coordinating aggregates, i.e., system modes for which particular

interaction rules hold.

Expressiveness: This is inherited from BIP as the possibility to directly specify any kind of static coordination without modifying the involved components or adding extra coordinating components. Regarding dynamic coordination, the proposed language directly encompasses the identified levels of dynamicity by supporting typing and the expressive power of first order logic. Nonetheless, explicit handling of quantifiers is limited to declarations that link variable names to actual instances.

Abstract Semantics: The language relies on an operational semantics that admits a variety of implementations between two extreme cases. One consists in pre-computing a global interaction constraint applied to an unstructured set of component instances and choosing the enabled interactions and the corresponding operations for a given configuration. The other consists in computing separately interactions for aggregates of components and combining them.

DReAM differentiates from L-DReAM by changing two of its core concepts: atomic components and compounds. The behavior of an atomic component in DReAM is explicitly defined as a labeled transition system instead of being encoded using the core coordination language, which makes the definition of multi-stage automatons more intuitive. Motifs are obtained from L-DReAM compounds by stripping them of their interface and local variables while equipping them with a map to parametrize component coordination and allowing to access utility functions to simplify map access and management. These differences are the result of specific choices tailored to the classes of actual systems and use cases that the framework aims at modeling, while also allowing concrete implementations that are more resource-efficient. These include adaptive systems, where the focus is not in the ability of capturing precisely how all the internals work but rather how their macroscopic behavior needs to react and adapt to changes in the environment, or collaborative mobile systems, where the focus is to study and guide emerging collective behavior while varying the topology and coordination rules underneath.

Lastly, we provided an experimental validation of the theoretical frameworks by presenting the prototype implementation jDReAM and

studying how it handles some simple yet relevant examples modeled using DReAM. While at this maturity stage jDReAM is meant to represent only a "proof of concept" of an executable environment, there are many opportunities for improvement and extension of its features that make it a viable ground for future development.

6.2 Related works

L-DReAM and DReAM allow both conjunctive and disjunctive style modeling of dynamic reconfigurable systems. They inherit the expressiveness of the coordination mechanisms of BIP [11] as they directly encompass multiparty interaction and extend previous work on modeling parametric architectures in Dy-BIP [12] in many respects. In both frameworks interactions involve not only transferal of values but also encompass reconfiguration. DReAM takes this one step further and supports self-organization "out of the box" by relying on the notions of maps and motifs.

BIP is a mature framework that leverages *connectors* to model interactions among components and define the computation flow between them. System definitions can be parametric, but connectors remain essentially "static" and defined from a somewhat "global" perspective by matching ports from different components. This restriction allows BIP to synthesize very efficient executable runtimes from system specifications, but prevents it from modeling dynamic systems effectively.

Dy-BIP was introduced with the intent of addressing the efficient specification of dynamic systems sharing some ground theory and terminology with BIP. It uses previous work on the encoding of BIP connectors with PIL, but instead of specifying them from a global perspective it adopts a more "compositional" approach by attaching *interaction constraints* to the ports of components. From these, a *global interaction constraint* is built at each system state as the conjunction of individual constraints, and the interaction that is performed is a solution to it.

L-DReAM and DReAM adopt a coordination language that, just like Dy-BIP, is built on the foundation of PIL, but gives the designer more freedom on how to use it to the point of allowing any style between two

"extremes" that we referred to as *disjunctive* and *conjunctive*.

When the disjunctive style is adopted, both frameworks can be considered as exogenous coordination languages, e.g., like an ADL. To the best of our knowledge L-DReAM and DReAM surpass existing exogenous coordination frameworks in that hey offer a well-thought and methodologically complete set of primitives and concepts to model parametric, dynamic and reconfigurable system architectures.

When the conjunctive style is adopted, they can be used as endogenous coordination languages comparable to process calculi to the extent that they rely on a single associative parallel composition operator. In DReAM and L-DReAM this operator is logical conjunction. It is easy to show that for existing process calculi parallel composition is a specialization of conjunction in Interaction Logic. For CCS [40] the causal rules are of the form $p \Rightarrow \mathbf{true} \wedge \bar{p} \Rightarrow \mathbf{true}$, where p and $\bar{p}$ are input and output port names corresponding to port symbol p. In this context, strong synchronization can also be modeled without resorting to restriction by using causal rules like $p \Rightarrow \bar{p} \wedge \bar{p} \Rightarrow p$. For CSP [14], the interface parallel operator parametrized by the shared channel a can be modeled in PIL by defining a set of ports A implementing a and using causal rules of the form $a_i \Rightarrow \bigwedge_{a_j \in A} a_j$ for all $a_i \in A$.

We have briefly hinted at how we can model some aspects of other richer calculi that offer the possibility of modeling dynamic infrastructures via channel passing, such as π-calculus [41], using L-DReAM with its reconfiguration operations. The same applies for formalisms with richer communication models, such as AbC [1], offering multicast communications by selecting groups of partners according to predicates over their attributes. Attribute-based interaction can be simulated by our interaction mechanism involving guards on the exchanged values and atomic transfer of values.

Both frameworks were designed with autonomy in mind, especially DReAM. As such, it has some similarities with languages for autonomous systems, in particular robotic systems such as Buzz [50, 51]. Nonetheless, we believe that the presented frameworks are more general as they do not rely on assumptions about the timed synchronous cyclic behavior of

components.

Finally, DReAM shares the same conceptual framework with DR-BIP [7]. The latter is an extension of BIP with component dynamism and reconfiguration. As such it adopts an exogenous and imperative approach based on the use of connectors.

6.3 Future work

The presented frameworks have been developed with due attention to sound formal definitions of their core concepts and empirically validated through the implementation of an execution engine paired with libraries that support the translation of L-DReAM and DReAM specifications to Java. These two tracks still require further work for the frameworks to reach maturity.

From the theoretical standpoint, early intuitions on the relationship between conjunctive and disjunctive style need to be investigated in order to formalize under which conditions disjunctive rules can be converted to conjunctive ones. Furthermore, while the language adopted to define coordination rules allows to enforce properties on interactions by construction, the formal approach adopted by the framework enables the use of (possibly automated) analysis techniques that would also allow verification of structural and behavioral properties of specifications. The development of such techniques is a key step to take full advantage of the frameworks and support their actual application. The capability of L-DReAM to capture and model several elements of other process description languages hinted in section 3.3 can also benefit from a more through selection of comparable languages to define sound transformations from one to the other, not only to test the ability of the framework to capture different paradigms but also to verify to what extent it is possible to map them in a taxonomy of coordination approaches induced by the dichotomy of styles "conjunctive vs disjunctive". Further research questions are concerned with the study of congruence and similarity relations for L-DReAM and DReAM systems.

From the implementation standpoint, the reworked jDReAM library

offers many opportunities for improvement and extension. From a usability perspective, the framework would greatly benefit from an integrated development environment capable of compiling a domain specific language closer to the theoretical syntax of L-DReAM and DReAM into Java code that uses the jDReAM API. This environment should also be enriched with plug-ins to monitor the jDReAM execution engine and simulate the execution of complex systems with convenient graphical interfaces. Under the hood, there are several optimizations and further developments that can be done to improve the performance of the engine. One of these is to enable support for multi-processing by allowing compounds and motifs to have dedicated execution engines running on different processes (and possibly different machines), organizing them in hierarchies matching the system structure with parent processes collecting the computation of their descendants. Having the possibility of turning on and off this feature would enable us to exploit parallelism to divide the problem of computing the global coordination constraint into smaller problems for complex systems with complicated structures without suffering from the unneeded synchronization overhead when dealing with "flatter" systems that cannot benefit as much from parallelization. Lastly, as the coordination language is fully represented with dedicated classes implementing the syntax of the interaction logic, developing a translator to propositional logic over port names could allow the integration of third-party state-of-the-art solvers to increase the performance of the engine in finding solutions to the global coordination constraints.

Appendix A

Proofs

A.1 Disjunctive to conjunctive transformation in PIL

A global disjunctive constraint Ψ can be transformed into the conjunction of a series of conjunctive constraints of the form $p \Rightarrow \Psi_p$ *if and only if* $\emptyset \models \Psi$.

Proof. Let P be the set of ports over which the formula Ψ is defined.

The necessity of $\emptyset$ being a model of Ψ in order to have a corresponding conjunctive formulation is trivial, as $\emptyset$ is always a model for any conjunctive formula $\bigwedge_{p \in P} p \Rightarrow \Psi_p$, and two PIL formulas are equivalent if and only if they are satisfied by the same interactions.

To prove the condition is also sufficient, we first observe that

$$\emptyset \models \Psi \quad \Rightarrow \quad \Psi \equiv \Psi \vee \bigwedge_{p \in P'} \neg p \tag{A.1}$$

where $P' \subseteq P$.

Let us write Ψ in the disjunctive normal form:

$$\Psi = \bigvee_{i=1}^{n} \left(\bigwedge_{p_i \in P_i} p_i \wedge \bigwedge_{q_i \in Q_i} \neg q_i \right) \tag{A.2}$$

where $P_i, Q_i \subseteq P$.

From (A.1), it follows that we can use the logical disjunction to combine Ψ with any monomial consisting only of inhibited ports. In particular, we can consider an equivalent formula Ψ' defined in the following way:

$$\Psi' = \Psi \vee \bigwedge_{p \in P} \neg p \tag{A.3}$$

By applying De Morgan's law, we can transform Ψ' in the form:

$$\begin{aligned} \Psi' = & \bigvee_{i=1}^{n} \left(\bigwedge_{p_i \in P_i} p_i \wedge \bigwedge_{q_i \in Q_i} \neg q_i \right) \vee \bigwedge_{p \in P} \neg p = \\ = & \overline{\bigwedge_{i=1}^{n} \left(\overline{\bigwedge_{p_i \in P_i} p_i \wedge \bigwedge_{q_i \in Q_i} \neg q_i} \right) \wedge \bigvee_{p \in P} p} = \\ = & \overline{\bigwedge_{i=1}^{n} \left(\bigvee_{p_i \in P_i} \neg p_i \vee \bigvee_{q_i \in Q_i} q_i \right) \wedge \bigvee_{p \in P} p} \end{aligned} \tag{A.4}$$

From here, we can apply the distributivity property of the logical conjunction obtaining:

$$\begin{aligned} & \overline{\bigwedge_{i=1}^{n} \left(\bigvee_{p_i \in P_i} \neg p_i \vee \bigvee_{q_i \in Q_i} q_i \right) \wedge \bigvee_{p \in P} p} = \\ = & \overline{\bigvee_{\substack{p \in P \\ j \in J \\ i \in I}} \left(\bigwedge_{p_i \in P_i} \neg p_i \wedge \bigwedge_{q_j \in Q_j} q_j \wedge p \right)} \end{aligned} \tag{A.5}$$

where $I \cup J = [1..n]$ and $I \cap J = \emptyset$.

By applying again De Morgan's law until we remove the negations we

obtain the conjunctive normal form:

$$\begin{aligned}
&\overline{\bigvee_{\substack{p\in P\\I,J}}\left(\bigwedge_{p_i\in P_i}\neg p_i\wedge\bigwedge_{q_j\in Q_j}q_j\wedge p\right)}=\\
&=\bigwedge_{\substack{p\in P\\I,J}}\left(\overline{\bigwedge_{p_i\in P_i}\neg p_i\wedge\bigwedge_{q_j\in Q_j}q_j\wedge p}\right)=\\
&=\bigwedge_{\substack{p\in P\\I,J}}\left(\bigvee_{p_i\in P_i}p_i\vee\bigvee_{q_j\in Q_j}\neg q_j\vee\neg p\right)=\\
&=\bigwedge_{\substack{p\in P\\I,J}}p\Rightarrow\left(\bigvee_{p_i\in P_i}p_i\vee\bigvee_{q_j\in Q_j}\neg q_j\right)
\end{aligned}\tag{A.6}$$

which is indeed an equivalent conjunctive-style formula. □

Appendix B

jDReAM code examples

B.1 Coordinating flocks of interacting robots

The following code snippets show one possible way of implementing the component types described in section 4.3.1 using the jDReAM libraries.

We start with the definition of the `Robot` class, which implements the component type by the same name extending the `AbstractComponent` class. The constructor is parametrized with the parent `Entity` that will host it in its pool (i.e., the root motif of the example), the sensor's range and the robot's initial direction.

```
public class Robot extends AbstractComponent {

  public Robot(Entity parent,double range,int dirx,int diry) {
    super(parent);

    setInterface(new Port("tick",this));

    setStore(
        new VarStore(
            new LocalVariable("clock",new NumberValue(0)),
            new LocalVariable("range",new NumberValue(range)),
            new LocalVariable("ts",new NumberValue(0)),
            new LocalVariable("dir",new ArrayValue(dirx,diry))));

    setBehavior(newBehavior(getInterface(),getStore()));

  }

  ...
```

```
  ...

  private static LTS newBehavior(
      Map<String, Port> cInterface,
      VarStore store) {

    Map<ControlLocation, Set<Transition>> transitions = new HashMap<>();
    ControlLocation currentControlLocation = new ControlLocation("s0");

    transitions.put(currentControlLocation, new HashSet<>());
    transitions.get(currentControlLocation)
    .add(
        new Transition(
            currentControlLocation,
            new Term(
                new PortAtom(cInterface.get("tick")),
                new Assign(
                    new VariableActual(store.getLocalVariable("clock")),
                    new Sum(
                        new VariableActual(store.getLocalVariable("clock")),
                        new NumberValue(1)))),
            currentControlLocation));

    return new LTS(transitions, currentControlLocation);
  }
}
```

Next, we show the definition of the class implementing the motif coordinating all the robots. The constructor accepts as parameters the size of the square grid used to initialize the map, and the sensor's range for all `Robot` instances. An array of initial directions for each `Robot` instance is chosen and used to setup the initial configuration of the motif.

```
public class InteractiveRobots extends AbstractMotif {

  public InteractiveRobots(int size,double range) {
    super(new GridMap(size,size));

    int[][] dirs = {{0,1},{1,1},{1,0},{1,-1},{0,0},
       {0,-1},{-1,-1},{-1,0},{-1,1}};
    int[][] addrs = new int[9][2];
    int step = size/3;

    for (int i=0; i<3; i++) {
      for (int j=0; j<3; j++) {
        addrs[i*3+j][0] = step*j;
        addrs[i*3+j][1] = step*i;
      }
    }

    for (int i=0; i<9; i++) {
      Entity robot = new Robot(this,range,dirs[i][0],dirs[i][1]);
      addToPool(robot);
      setEntityPosition(robot,map.getNodeForAddress(new ArrayValue(addrs[i])));
    }

  ...
```

The specification is completed by defining the rules that coordinate the movement of robots and their direction update:

```
    ...

    EntityInstanceActual scope = new EntityInstanceActual(this);

    Declaration allRobots = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Robot.class));
    EntityInstanceRef c = allRobots.getVariable();

    Rule r1 = new FOILRule(allRobots,
        new ConjunctiveTerm(
            new PortReference(c,"tick"),
            Tautology.getInstance(),
            new Move(c,
                new MapNodeForAddress(scope,
                    new Sum(
                        new MapNodeAddress(new MapNodeForEntity(c)),
                        new VariableRef(c,"dir"))))));

    Declaration allRobots1 = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Robot.class));
    EntityInstanceRef c1 = allRobots1.getVariable();

    Declaration allRobots2 = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Robot.class));
    EntityInstanceRef c2 = allRobots2.getVariable();

    Rule r2 = new FOILRule(allRobots1,
        new FOILRule(allRobots2,
            new ConjunctiveTerm(
                new PortReference(c1,"tick"),
                new PortReference(c2,"tick"),
                new IfThenElse(
                    new And(
                        new LessThan(
                            new MapNodeDistance(
                                new MapNodeForEntity(c1),
                                new MapNodeForEntity(c2)),
                            new VariableRef(c1,"range")),
                        new Or(
                            new LessThan(
                                new VariableRef(c1,"ts"),
                                new VariableRef(c2,"ts")),
                            new And(
                                new Equals(
                                    new VariableRef(c1,"ts"),
                                    new VariableRef(c2,"ts")),
                                new LessThan(
                                    new InstanceIdentifier(c1),
                                    new InstanceIdentifier(c2))))),
                    new OperationsSequence(
                        new Assign(new VariableRef(c1,"dir"),
                            new VariableRef(c2,"dir")),
                        new Assign(new VariableRef(c1,"ts"),
                            new VariableRef(c1,"clock")))))));
    setRule(new AndRule(r1,r2));
  }
}
```

B.2 Simple platooning protocol for automated highways

The following code snippets show one possible way of implementing the component types described in section 4.3.4 using the jDReAM libraries.

First, we define a class implementing the component type *Car* by extending the `AbstractComponent` class:

```
public class Car extends AbstractComponent {

  public static NumberValue splitProb = new NumberValue(0.2);

  public Car(Entity parent,double position,double speed) {
    super(parent);

    setInterface(new Port("initSplit"),new Port("ackSplit"),
        new Port("closeSplit"),new Port("initJoin"),
        new Port("ackJoin"),new Port("finishJoin"));

    setStore(new VarStore(
        new LocalVariable("id",new NumberValue(id)),
        new LocalVariable("pos",new NumberValue(position)),
        new LocalVariable("speed",new NumberValue(speed))));

    setBehavior(newBehavior(getInterface(),getStore()));
  }

  private static LTS newBehavior(Map<String,Port> cInterface,VarStore store) {
    Map<ControlLocation,Set<Transition>> transitions = new HashMap<>();
    ControlLocation currentControlLocation = new ControlLocation("cruising");

    ControlLocation c1, c2;

    c1 = currentControlLocation;
    c2 = new ControlLocation("splitting");
    transitions.put(c1,new HashSet<>());
    transitions.get(c1).add(new Transition(c1,
        new Term(new And(
            new LessThan(new RandomNumber(),splitProb),
            new PortAtom(cInterface.get("initSplit")))),c2));
    transitions.get(c1).add(new Transition(c1,
        new Term(new PortAtom(cInterface.get("ackSplit"))),c2));
    transitions.put(c2,new HashSet<>());
    transitions.get(c2).add(new Transition(c2,
        new Term(new PortAtom(cInterface.get("closeSplit"))),c1));
    c2 = new ControlLocation("joining");
    transitions.put(c2,new HashSet<>());
    transitions.get(c1).add(new Transition(c1,
        new Term(new PortAtom(cInterface.get("initJoin"))),c2));
    transitions.get(c1).add(new Transition(c1,
        new Term(new PortAtom(cInterface.get("ackJoin"))),c2));
    transitions.get(c2).add(new Transition(c2,
        new Term(new PortAtom(cInterface.get("finishJoin"))),c1));

    return new LTS(transitions,currentControlLocation);
  }
}
```

Here you can see that the probability of initiating a splitting maneuver and the corresponding condition on the port *initSplit* are handled by the `Car` components themselves instead of leaving them to the *Platoon* motifs as in sub-rule (4.27) of r_P. The latter is instead defined by declaring a `Platoon` class extending the `AbstractMotif` class.

```
public class Platoon extends AbstractMotif {

  public Platoon(Entity parent,Entity[] initialPool) {
    super(parent,
        Arrays.stream(initialPool).collect(Collectors.toSet()),
        new ArrayMap(initialPool.length));
    map.setOwner(this);

    for (int i=0; i<initialPool.length; i++)
      setEntityPosition(initialPool[i],((ArrayMap)map).getNodeAtIndex(i));

    setRule(newRule(this));
  }

  @Override
  public MapNode createMapNode() {
    MapNode newNode = super.createMapNode();
    newNode.getStore().setVarValue("newLeader",new NumberValue(-1));
    newNode.getStore().setVarValue("newLoc",new NumberValue(-1));
    return newNode;
  }

  ...
```

Notice that we are overriding the `createMapNode` method of the superclass in order to equip every location in the associated map with the local variables *newLeader* and *newLoc*. The creation of the rule coordinating the components in the motif is delegated to the static method `newRule`, which returns a `Rule` object directly implementing (4.26-4.32):

```
  ...

  private static Rule newRule(AbstractMotif current) {
    EntityInstanceActual scope = new EntityInstanceActual(current);

    Declaration cars = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Car.class));
    EntityInstanceRef c = cars.getVariable();

    Rule r1 = new FOILRule(cars,
        new Term(new Assign(new VariableRef(c,"pos"),
            new Sum(
                new VariableRef(c,"pos"),
                new VariableRef(c,"speed")))));

  ...
```

```
...

cars = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
c = cars.getVariable();

Rule r2 = new FOILRule(cars,
    new ConjunctiveTerm(
        new PortReference(c,"initSplit"),
        new And(
            new GreaterThan(new PoolSize(scope),new NumberValue(1)),
            new Not(new Equals(
                new VariableRef(new MapPropertyRef<>(scope,"head"),"id"),
                new VariableRef(c,"id"))))));

Declaration cars1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
var c1 = cars1.getVariable();
Declaration cars2 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
var c2 = cars2.getVariable();

Rule r3 = new FOILRule(cars1,
    new FOILRule(cars2,
        new ConjunctiveTerm(
            new PortReference(c1,"ackJoin"),
            new PortReference(c2,"ackJoin"))));

cars1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
c1 = cars1.getVariable();
cars2 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
c2 = cars2.getVariable();

Rule r4 = new FOILRule(cars1,
    new FOILRule(cars2,
        new ConjunctiveTerm(
            new PortReference(c1,"initJoin"),
            new PortReference(c2,"initJoin"))));

cars1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
c1 = cars1.getVariable();
cars2 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Car.class));
c2 = cars2.getVariable();
Rule r5 = new FOILRule(cars1,
    new FOILRule(cars2,
        new ConjunctiveTerm(
            new PortReference(c1,"finishJoin"),
            new PortReference(c2,"finishJoin"))));

...
```

```
    ...

    cars1 = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Car.class));
    c1 = cars1.getVariable();
    cars2 = new Declaration(
        Quantifier.EXISTS, scope, new TypeRestriction(Car.class));
    c2 = cars2.getVariable();

    Rule r6 = new FOILRule(cars1,
        new FOILRule(cars2,
            new ConjunctiveTerm(
                new PortReference(c1,"ackSplit"),
                new PortReference(c2,"initSplit"))));

    cars1 = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Car.class));
    c1 = cars1.getVariable();
    cars2 = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Car.class));
    c2 = cars2.getVariable();

    Rule r7 = new FOILRule(cars1,
        new FOILRule(cars2,
            new ConjunctiveTerm(
                new PortReference(c1,"initSplit"),
                new Or(
                    new SameInstance(c1,c2),
                    new PortReference(c2,"ackSplit")))));

    return new AndRule(r1,r2,r3,r4,r5,r6,r7);
  }
}
```

Lastly, the *Road* motif type is also implemented by defining a sub-class of `AbstractMotif` with a rule implementing r_R:

```
public class Road extends AbstractMotif {

  public Value speedUp;
  public Value speedDown;

  public Road(Set<Entity> pool,double joinDistance,double speedDelta) {
    super(null,pool,new DummyMap());
    map.setOwner(this);
    MapNode node = map.getNodes().stream().findFirst().get();
    pool.stream().forEach(e -> map.setEntityMapping(e,node));
    speedUp = new NumberValue(1+speedDelta);
    speedDown = new NumberValue(1-speedDelta);

    EntityInstanceActual scope = new EntityInstanceActual(this);
    Declaration platoons1 = new Declaration(
        Quantifier.FORALL, scope, new TypeRestriction(Platoon.class));
    EntityInstanceRef p1 = platoons1.getVariable();
    Declaration cars = new Declaration(
        Quantifier.FORALL, p1, new TypeRestriction(Car.class));
    EntityInstanceRef c = cars.getVariable();
    Declaration platoons2 = new Declaration(
        Quantifier.EXISTS, scope, new TypeRestriction(Platoon.class));
    EntityInstanceRef p2 = platoons2.getVariable();

    ...
```

```
...

MapNodeRef n = new MapNodeRef();

Rule join1 = new FOILRule(platoons1,
    new FOILRule(cars,
        new FOILRule(platoons2,
            new ConjunctiveTerm(
                new PortReference(c,"initJoin"),
                new And(
                    new Not(new SameInstance(p1,p2)),
                    new LessThan(
                        new NumberValue(0),
                        new Difference(
                            new VariableRef(
                                new MapPropertyRef<>(p2,"tail"),"pos"),
                            new VariableRef(
                                new MapPropertyRef<>(p1,"head"),"pos")),
                        new NumberValue(joinDistance)),
                    new PortReference(
                        new MapPropertyRef<>(p2,"head"),"ackJoin")),
                new CreateMapNode(p2,n,
                    new OperationsSequence(
                        new Assign(new VariableRef(c,"speed"),
                            new VariableRef(
                                new MapPropertyRef<>(p2,"head"),"speed")),
                        new Assign(
                            new VariableRef(
                                new MapNodeForEntity(c),"newLeader"),
                            new VariableRef(
                                new MapPropertyRef<>(p2,"head"),"id")),
                        new Assign(
                            new VariableRef(new MapNodeForEntity(c),"newLoc"),
                            new Sum(
                                new MapSize(p2),
                                new VariableRef(
                                    new MapNodeForEntity(c),"index")))))))));

platoons1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Platoon.class));
p1 = platoons1.getVariable();
platoons2 = new Declaration(
    Quantifier.EXISTS, scope, new TypeRestriction(Platoon.class));
p2 = platoons2.getVariable();

Rule join1b = new FOILRule(platoons1,
    new FOILRule(platoons2,
        new ConjunctiveTerm(
            new PortReference(new MapPropertyRef<>(p1,"head"),"ackJoin"),
            new And(
                new PortReference(new MapPropertyRef<>(p2,"head"),"initJoin"),
                new LessThan(
                    new NumberValue(0),
                    new Difference(
                        new VariableRef(
                            new MapPropertyRef<>(p1,"tail"),"pos"),
                        new VariableRef(
                            new MapPropertyRef<>(p2,"head"),"pos")),
                    new NumberValue(joinDistance))))));
...
```

```
...

platoons1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Platoon.class));
p1 = platoons1.getVariable();
cars = new Declaration(
    Quantifier.FORALL, p1, new TypeRestriction(Car.class));
c = cars.getVariable();
platoons2 = new Declaration(
    Quantifier.EXISTS, scope, new TypeRestriction(Platoon.class));
p2 = platoons2.getVariable();

Rule join2 = new FOILRule(platoons1,
    new FOILRule(cars,
        new FOILRule(platoons2,
            new ConjunctiveTerm(
                new PortReference(c,"finishJoin"),
                new And(
                    new PortReference(
                        new MapPropertyRef<>(p2,"head"),"finishJoin"),
                    new Or(
                        new Equals(
                            new VariableRef(
                                new MapPropertyRef<>(p2,"head"),"id"),
                            new VariableRef(
                                new MapNodeForEntity(c),"newLeader")),
                        new SameInstance(p1,p2))),
                new IfThenElse(
                    new Not(new SameInstance(p1,p2)),
                    new OperationsSequence(
                        new MigrateMotif(c,p2,
                            new MapNodeVarEquals(p2,"index",new VariableRef(
                                new MapNodeForEntity(c),"newLoc"))),
                        new DeleteInstance(p1)))))));

platoons1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Platoon.class));
p1 = platoons1.getVariable();
cars = new Declaration(
    Quantifier.FORALL, p1, new TypeRestriction(Car.class));
c = cars.getVariable();
p2 = new EntityInstanceRef();

n = new MapNodeRef();
VariableRef i = new VariableRef("i");

Rule split1 = new FOILRule(platoons1,
    new FOILRule(cars,
        new ConjunctiveTerm(
            new PortReference(c,"initSplit"),
            new OperationsSequence(
                new Assign(
                    new VariableRef(c,"speed"),
                    new Product(new VariableRef(c,"speed"),speedDown)),
                new CreateMotifInstance(Platoon.class,scope,
                    new MapNodeActual(((DummyMap)this.map).getNode()),p2,
                    new OperationsSequence(
                        new CreateMapNode(p2,n,
                            new MigrateMotif(c,p2,n)),
...
```

```
...
                        new ForLoop(i,
                            new Sum(
                                new NumberValue(1),
                                new VariableRef(
                                    new MapNodeForEntity(c),"index")),
                            new MapSize(p1),
                            new CreateMapNode(p2)))),
                new DeleteMapNode(new MapNodeForEntity(c))))));

platoons1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Platoon.class));
p1 = platoons1.getVariable();
Declaration cars1 = new Declaration(
    Quantifier.FORALL, p1, new TypeRestriction(Car.class));
EntityInstanceRef c1 = cars1.getVariable();
Declaration cars2 = new Declaration(
    Quantifier.EXISTS, p1, new TypeRestriction(Car.class));
EntityInstanceRef c2 = cars2.getVariable();

Rule split2 = new FOILRule(platoons1,
    new FOILRule(cars1,
        new FOILRule(cars2,
            new ConjunctiveTerm(
                new PortReference(c1,"ackSplit"),
                new PortReference(c2,"initSplit"),
                new IfThenElse(
                    new GreaterThan(
                        new VariableRef(new MapNodeForEntity(c1),"index"),
                        new VariableRef(new MapNodeForEntity(c2),"index")),
                    new OperationsSequence(
                        new Assign(
                            new VariableRef(
                                new MapNodeForEntity(c1),"newLeader"),
                            new VariableRef(c2,"id")),
                        new Assign(
                            new VariableRef(
                                new MapNodeForEntity(c1),"newLoc"),
                            new Difference(
                                new VariableRef(
                                    new MapNodeForEntity(c1),"index"),
                                new VariableRef(
                                    new MapNodeForEntity(c2),"index"))),
                        new Assign(
                            new VariableRef(c1,"speed"),
                            new Product(
                                new VariableRef(c1,"speed"),speedDown))),
                    new Assign(
                        new VariableRef(c1,"speed"),
                        new Product(new VariableRef(c1,"speed"),speedUp))
                    ))));

platoons1 = new Declaration(
    Quantifier.FORALL, scope, new TypeRestriction(Platoon.class));
p1 = platoons1.getVariable();
cars = new Declaration(
    Quantifier.FORALL, p1, new TypeRestriction(Car.class));
c = cars.getVariable();

...
```

```
    ...

    platoons2 = new Declaration(
        Quantifier.EXISTS, scope, new TypeRestriction(Platoon.class));
    p2 = platoons2.getVariable();

    Rule split3 = new FOILRule(platoons1,
        new FOILRule(cars,
            new FOILRule(platoons2,
                new ConjunctiveTerm(
                    new PortReference(c,"closeSplit"),
                    new Or(
                        new And(
                            new Equals(
                                new VariableRef(
                                    new MapNodeForEntity(c),"newLeader"),
                                new VariableRef(
                                    new MapPropertyRef<>(p2,"head"),"id")),
                            new PortReference(
                                new MapPropertyRef<>(p2,"head"),"closeSplit")),
                        new Equals(
                            new VariableRef(
                                new MapNodeForEntity(c),"newLeader"),
                            new NumberValue(-1))),
                    new IfThenElse(
                        new Not(
                            new Equals(
                                new VariableRef(
                                    new MapNodeForEntity(c),"newLeader"),
                                new NumberValue(-1))),
                        new OperationsSequence(
                            new MigrateMotif(c,p2,
                                new MapNodeVarEquals(p2,"index",
                                    new VariableRef(
                                        new MapNodeForEntity(c),"newLoc"))),
                            new DeleteMapNode(new MapNodeForEntity(c))))))));

    setRule(new AndRule(join1,join1b,join2,split1,split2,split3));

  }
}
```

The tests ran in section 5.3.3 and presented in figure 41 have been generated and executed by running the jDReAM `ExecutionEngine` as follows:

```
Platooning road = generateConfiguration(pN[i],cN[i],avgS[i],avgSVar[i],
    avgCS[i],avgCSVar[i],avgPS[i],avgPSVar[i],jD[i],sD[i]);

String output = String.format(
          "platooning_%d_%d_%.2f_%.2f_%.2f_%.2f_%.2f_%.2f_%.2f_%.2f_%dc.json",
          pN[i],cN[i],avgS[i],avgSVar[i],avgCS[i],avgCSVar[i],avgPS[i],
          avgPSVar[i],jD[i],sD[i],cycles[i]);

ExecutionEngine ex = new ExecutionEngine(
    road,GreedyStrategy.getInstance(),new ConsoleOutput(),100,output);

ex.run();
```

Bibliography

[1] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. "On the Power of Attribute-Based Communication". In: *Proceedings of Formal Techniques for Distributed Objects, Components, and Systems - FORTE 2016 - 36th IFIP WG 6.1 In'l Conference*. 2016, pp. 1–18.

[2] Yehia Abd Alrahman et al. "A calculus for attribute-based communication". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. ACM, 2015, pp. 1840–1845.

[3] Luca Aceto et al. *Reactive systems: modelling, specification and verification*. cambridge university press, 2007.

[4] Robert Allen and David Garlan. "A formal basis for architectural connection". In: *ACM Transactions on Software Engineering and Methodology* (1997).

[5] Farhad Arbab. "Reo: a channel-based coordination model for component composition". In: *Math. Struct. Comput. Sci.* 14.3 (2004), pp. 329–366.

[6] Paul Attie et al. "A general framework for architecture composability". In: *Software Engineering and Formal Methods*. Springer, 2014, pp. 128–143.

[7] Rim El Ballouli et al. "Four Exercises in Programming Dynamic Reconfigurable Systems: Methodology and Solution in DR-BIP". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods*. Vol. 11246. Lecture Notes in Computer Science. Springer, 2018, pp. 304–320.

[8] Patrick J. Barnes et al. *Prototyping hard real-time Ada systems in a classroom environment*. Tech. rep. 1992.

[9] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling heterogeneous real-time components in BIP". In: *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods*. IEEE. 2006, pp. 3–12.

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

[11] Simon Bliudze and Joseph Sifakis. "The algebra of connectors - structuring interaction in BIP". In: *IEEE Transactions on Computers* 57.10 (2008), pp. 1315–1330.

[12] Marius Bozga et al. "Modeling dynamic architectures using Dy-BIP". In: *Proceedings of the 11th International Conference on Software Composition*. Springer. 2012, pp. 1–16.

[13] Jeremy S. Bradbury. "Organizing definitions and formalisms for dynamic software architectures". In: *Technical Report* 477 (2004).

[14] Stephen D. Brookes, Charles A. R. Hoare, and Andrew W. Roscoe. "A theory of communicating sequential processes". In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 560–599.

[15] Roberto Bruni et al. "A Formal Support to Business and Architectural Design for Service-Oriented Systems". In: *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*. Vol. 6582. Lecture Notes in Computer Science. Springer, 2011, pp. 133–152.

[16] Arvid Butting et al. "A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages". In: *Proceedings of the 4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering*. 2017, p. 13.

[17] Paul C. Clements. "A survey of architecture description languages". In: *Proceedings of the 8th international workshop on software specification and design*. IEEE Computer Society. 1996, p. 16.

[18] Rocco De Nicola, Alessandro Maggi, and Joseph Sifakis. "The DReAM framework for dynamic reconfigurable architecture modelling: theory and applications". In: *International Journal on Software Tools for Technology Transfer* (2020), pp. 1–19.

[19] Stephen H. Edwards et al. “Part II: Specifying components in RESOLVE”. In: *ACM SIGSOFT Software Engineering Notes* 19.4 (1994), pp. 29–39.

[20] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The architecture analysis & design language (AADL): An introduction*. Tech. rep. DTIC Document, 2006.

[21] José Luiz Fiadeiro and T. S. E. Maibaum. “Categorical Semantics of Parallel Program Design”. In: *Sci. Comput. Program.* 28.2-3 (1997), pp. 111–138.

[22] Ariel D. Fuxman. “A survey of architecture description languages”. In: *Reports from CSC2108 Automatic Verification* (2000).

[23] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[24] David Garlan. “Software architecture: a travelogue”. In: *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 29–39.

[25] David Garlan and Mary Shaw. “An introduction to software architecture”. In: *Advances in Software Engineering and Knowledge Engineering*. Ed. by V. Ambriola and G. Tortora. Vol. I. World Scientific Publishing Company, New Jersey, 1993.

[26] Stephen Gilmore and Jane Hillston. “The PEPA workbench: A tool to support a process algebra-based approach to performance modelling”. In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 1994, pp. 353–368.

[27] Gregor Gößler and Joseph Sifakis. “Composition for Component-Based Modeling”. In: *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*. Vol. 2852. Lecture Notes in Computer Science. Springer, 2002, pp. 443–466.

[28] Charles A. R. Hoare. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[29] Charles A. R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[30] Ann Hsu et al. *Design of platoon maneuver protocols for IVHS*. UC Berkeley: California Partners for Advanced Transportation Technology, 1991.

[31] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. "Middleware - layer connector synthesis: Beyond state of the art in middleware interoperability". In: *Formal Methods for Eternal Networked Software Systems*. Springer, 2011, pp. 217–255.

[32] James Ivers et al. *Documenting component and connector views with UML 2.0*. Tech. rep. DTIC Document, 2004.

[33] Daniel Jackson. "Alloy: a lightweight object modelling notation". In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (2002), pp. 256–290.

[34] Farnam Jahanian and Aloysius K. Mok. "Modechart: A specification language for real-time systems". In: *IEEE Transactions on Software Engineering* 20.12 (1994), pp. 933–947.

[35] David C. Luckham et al. "Partial orderings of event sets and their application to prototyping concurrent, timed systems". In: *Journal of systems and Software* 21.3 (1993), pp. 253–265.

[36] Jeff Magee and Jeff Kramer. "Dynamic structure in software architectures". In: *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 1996.

[37] Alessandro Maggi, Rocco De Nicola, and Joseph Sifakis. "A Logic-Inspired Approach to Reconfigurable System Modelling". In: *From Reactive Systems to Cyber-Physical Systems*. Vol. 11500. Lecture Notes in Computer Science. Springer, 2019, pp. 181–201.

[38] Ivano Malavolta et al. "What industry needs from architectural languages: A survey". In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 869–891.

[39] Nenad Medvidovic, Eric M Dashofy, and Richard N Taylor. "Moving architectural description from under the technology lamppost". In: *Information and Software Technology* 49.1 (2007), pp. 12–31.

[40] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980.

[41] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, I". In: *Information and computation* 100.1 (1992), pp. 1–40.

[42] Peter Newton and James C. Browne. "The CODE 2.0 graphical parallel programming language". In: *Proceedings of the 6th international conference on Supercomputing*. ACM. 1992, pp. 167–177.

[43] Rocco De Nicola, Alessandro Maggi, and Joseph Sifakis. "DReAM: Dynamic Reconfigurable Architecture Modeling". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods*. Vol. 11246. Lecture Notes in Computer Science. Springer, 2018, pp. 13–31.

[44] Xavier Nicollin and Joseph Sifakis. "An overview and synthesis on timed process algebras". In: *International Conference on Computer Aided Verification*. Springer. 1991, pp. 376–398.

[45] Peyman Oreizy et al. "Issues in modeling and analyzing dynamic software architectures". In: *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*. 1998, pp. 54–57.

[46] Mourad Oussalah, Adel Smeda, and Tahar Khammaci. "An Explicit Definition of Connectors for Component-Based Software Architecture". In: *Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems*. IEEE, 2004, pp. 44–51.

[47] Mert Ozkaya and Christos Kloukinas. "Are we there yet? Analyzing architecture description languages for formal analysis, usability, and realizability". In: *Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2013, pp. 177–184.

[48] Jens Palsberg, Cun Xiao, and Karl Lieberherr. "Efficient implementation of adaptive software". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.2 (1995), pp. 264–292.

[49] Jennifer Pérez. "PRISMA: Aspect-Oriented Software Architectures". PhD thesis. Department of Information Systems and Computation, Polytechnic University of Valencia, 2006.

[50] Carlo Pinciroli and Giovanni Beltrame. "Buzz: An extensible programming language for heterogeneous swarm robotics". In: *Proceedings of 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2016, pp. 3794–3800.

[51] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. "Buzz: An extensible programming language for self-organizing heterogeneous robot swarms". In: *arXiv preprint arXiv:1507.05946* (2015).

[52] Frantisek Plasil and Stanislav Visnovsky. "Behavior protocols for software components". In: *Software Engineering, IEEE Transactions on* 28.11 (2002), pp. 1056–1076.

[53] Carlos E. C. Quintero et al. “Coordination in a reflective architecture description language”. In: *Proceedings of the 5th International Conference on Coordination Models and Languages*. Vol. 2315. Lecture Notes in Computer Science. Springer, 2002, pp. 141–148.

[54] Mary Shaw et al. “Abstractions for Software Architecture and Tools to Support Them”. In: *IEEE Transactions on Software Engineering* 21 (1995), pp. 314–335.

[55] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. “Liquid software manifesto: the era of multiple device ownership and its implications for software architecture”. In: *Proceedings of the 38th Computer Software and Applications Conference*. IEEE. 2014, pp. 338–343.

[56] Allan Terry et al. “Overview of Teknowledge’s domain-specific software architecture program”. In: *ACM SIGSOFT Software Engineering Notes* 19.4 (1994), pp. 68–76.

[57] Alexander L. Wolf et al. “Foundations for the Study of Software Architecture”. In: *ACM SIGSOFT Software Engineering Notes* 17 (1992), pp. 40–52.

[58] Chen Yang, Peng Liang, and Paris Avgeriou. “A systematic mapping study on the combination of software architecture and agile development”. In: *Journal of Systems and Software* 111 (2016), pp. 157–184.

Unless otherwise expressly stated, all original material of whatever nature created by Alessandro Maggi and included in this thesis, is licensed under a Creative Commons Attribution Noncommercial Share Alike 3.0 Italy License.

Check creativecommons.org/licenses/by-nc-sa/3.0/it/ for the legal code of the full license.

Ask the author about other uses.