SAPIENZA
UNIVERSITÀ DI ROMA

Verification of Artifact-centric Processes

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
ANTONIO RUBERTI

Dottorato di Ricerca in Ingegneria Informatica – XXV Ciclo

Candidate

Riccardo De Masellis
ID number 802280

Thesis Advisor

Prof. Giuseppe De Giacomo

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Ingegneria Informatica

March 25, 2013

Thesis defended on October 7, 2013
in front of a Board of Examiners composed by:

Prof. Andrea Clementi (chairman)
Prof. Paolo Ferragina
Prof. Roberto Basili

Verification of Artifact-centric Processes
Ph.D. thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Website: http://www.dis.uniroma1.it/~demasellis
Author's email: demasellis@dis.uniroma1.it

A mamma e papà.

Acknowledgments

I would like to express my sincere and genuine gratitude to my advisor professor Giuseppe De Giacomo, whose patience, competence, care and immense knowledge have been a guiding light and they always will be.

Unique thanks go to my colleagues, for creating an amazing work environment which I am proud to join every morning.

I am grateful to professors Alin Deutsch and Alessio Lomuscio for patiently and thoroughly reviewing my thesis and providing interesting comments and I would also like to thank professor Jianwen Su, whose kindness made one of the best experience of my life possible.

Abstract

An artifact-centric process is a process characterized by both static aspects, i.e., the data of interest, and dynamic aspects, i.e., its behavior. In this thesis we present a new framework for artifact-centric processes and we study conditions under which formal analysis is decidable. Indeed, such systems are, in general, unbounded, since they should explicitly model data and user inputs to be of practical interest. Our framework occupies the middle ground between reasoning about actions in artificial intelligence and the area of business process modeling and management, and has been also strongly influenced by literature on databases: data instances are the states of our process and they can change by means of actions, expressed in terms of preconditions and postconditions. We model possible external inputs as existential values, that, along with an acyclic condition on actions inspired by work on data exchange, guarantee the system to be bounded. This makes verification of temporal properties decidable. Temporal properties are expressed in a first-order variant of μ-calculus. The thesis starts from the work that has been the first to present such ideas, paving the way for future research, in which actions are based on conjunctive queries only (no negation). Then it shows a modification that allows for more expressive actions and an extension where a semantic layer is added to enrich the representation of information. In order to show that our approach is promising for practical application, a comparison with some other business process modeling formalisms is drawn. In the last part of the thesis we address the different, but related problem of runtime verification of artifact-centric processes.

Contents

Chapter 1

Introduction

A business process is a set of activities that realizes a specific goal. Since the industrial revolution, companies recognized that optimization and automation of processes play a fundamental role for both increasing productivity and saving costs. Indeed, not only material products, but more broadly, each product that a company produces, e.g., knowledge or expertise, is the result of a sequence of activities performed, hence of a process.

As technologies evolve and the complexity of processes grows, the need for techniques for efficient management of processes increases. In the last decades, the field of business process management (BPM) [138] addressed this issue and nowadays the off-the-shelf methodologies it offers are successfully used by companies in many practical cases.

The most glaring benefits deriving from BPM are automation, adaptation and analysis of processes. By a precise descriptions of tasks, resources, data involved in the process and their dependencies, it is possible to: *(i)* automatize non-human tasks with the help of proper software applications; *(ii)* adapt the process to react in real-time to environmental changes and *(iii)* check if the process satisfies high-level requirements.

More precisely, the role of BPM is the managing of business processes during their entire lifecycle. The lifecycle of a process can be roughly divided into four phases [138] [127]: in the *design and analysis* phase, starting from requirements, process schemas are designed using a suitable modeling language. Schemas can then be validated using verification techniques to discover errors and to check the correctness and consistency against the domain requirements. In the *configuration* phase process schemas are implemented by configuring a Process-Aware Information System (PAIS) in order to support process enactment via an execution engine. In the *enactment and monitoring* phase process instances are then initiated, executed and monitored by the run-time environment. The execution engine drives and monitors the work of the involved entities, and performed tasks generating execution traces are tracked and logged. After process execution, in the *diagnosis* phase process logs are evaluated and mined to identify problems and possible improvements, potentially resulting in process re-design and adaptation. These phases are not meant to be executed just once and for all, but rather multiple times circularly.

1.1 Process modeling

Business process modeling is a field where standardization efforts have essentially failed [127]. This can be mainly ascribed to the difficulty of capturing a broad range of requirements and purposes. Indeed, there are a plethora of models [1] for business processes that can be categorized according to different dimensions [138], such as degree of abstraction of business goals or degree of automaton. We divide models in literature according to their ability of supporting data.

Most BPM frameworks are organized around activity flows, (see, e.g., [133]) meaning that tasks and operations are the core building blocks of the model, whereas data manipulated by these processes is seen as a second-class citizen. Over the past several years, a data-centric approach has emerged [109, 81, 23, 44]. This approach focuses on augmented data records, known as *business artifacts* or simply *artifacts*, making them first-class citizens in the model. Artifacts are business-relevant objects that combine both data and process aspects into a holistic unit, and they are created, evolved, and (typically) archived as they pass through the business. An interesting analogy with programming languages can be drown: the same shift we experienced in the nineties, from functionals to object oriented programming, is now taking hold in the BPM community. Such an approach has several advantages. From a practical point of view, as happens with object oriented programming, we believe the artifact-centric approach to be more intuitive for business people. From a theoretical perspective, by combining data and processes, artifacts fill the gap between research on database and knowledge representation, that has focused largely on data aspects, and from research on programming languages, software engineering, workflow, and verification, that has centered mostly on process aspects.

In order to better understand the capabilities of data-centric approaches and differences between data-centric and activity centric frameworks, we now briefly survey the major models used in BPM. The main concepts of next two sections are taken from [107, 138, 127].

1.1.1 Activity-centric process models

Petri nets [130] are one of the oldest and best-known techniques for specifying business processes. The most important feature of Petri nets is their precise semantics that allows for formal analysis.

In their original formulation, Petri nets consist of places, transitions, and directed arcs connecting places and transitions. While the net's structure is fixed, places contain tokens that move to other adjacent places as a result of transitions. Given that transitions can change the state of a Petri net, i.e., the number and position of tokens, they intuitively represent active components such as events or tasks. Places, instead, being static components, are used to model states or conditions. Finally tokens represent physical or information objects.

Given that Petri nets are essentially a low-level formalism, they have been mostly used in control theory and robotics. Nonetheless several extensions of the original

[1] We adopt the following terminology: with *model* we refer to a language or general framework, e.g., the Petri nets model, while with *schema* we refer to an instance of the model, e.g., a specific Petri net, and finally with *instance* we mean a specific instantiation (or configuration) of the schema.

model has been proposed, allowing their use for a broader range of applications: one of such extensions is an attempt to inject data in the formalism. The resulting model, called *Colored Petri Nets* (CPNs) will be studied more in detail in Chapter 5.

Workflow nets [134] are an enhancement of Petri nets that allow for an easier representation of complex business processes by adding new concepts and notations. To give some examples, workflow nets can have a hierarchical structure and explicit split and join nodes to highlight exclusiveness and parallelism relationships between activities. Places represent conditions and tokens, by carrying information of specific processes, represent process instances. Nonetheless, data are not explicitly considered neither in the Petri net nor in the workflow net models: indeed, process execution is controlled by the control flow only and data-driven execution of any kind is not supported.

The *Business Process Modeling Notation* (BPMN) [111] is a graph-based modeling notation that has been developed under the coordination of the Object Management Group (OMG). As the Unified Modeling Language (UML) for object-oriented design and analysis, the scope of BPMN is to identify and combine in a single language the constructs and best practices of the other existing approaches at all level of abstraction, from business to technical level. It provides a wide range of modeling elements, from the basic ones, allowing to express simple structures, to complex ones, that provide more expressive power.

A BPMN diagram is formed by elements such as events, activities or data objects connected by relations like the control flow, the message flow and gateways.

Although the BPMN execution semantics is informally based on tokens, data impacts the process execution in terms of data-based decisions on gateways or as prerequisite of activities. However, data modeling is optional and this makes BPMN a model where data objects remain in the background, having a supporting role to the control flow.

The *Web-Services Business Process Execution Language* (BPEL) [110] has been proposed in 2003 by IT companies such as IBM, Microsoft and SAP, and it substantially merges the different approaches to specification of executable languages that such companies had. BPEL aims at defining a model that facilitates the integration of processes both within and between businesses. The interaction can be described as an *executable* business process, which describes the actual behavior of a participant in a business interaction, or as an *abstract* business process which, hiding some operational details for privacy, provides a sort of interface. It is a hierarchical block-structured language where each process is specified as a block. Dependencies between processes are captured by the so called control-links. BPEL schemas have the advantage of being executable but the language has no graphical representation as it is XML-based.

Data in BPEL is modeled through the use of variables that can be shared among the different participants in the process, and they essentially represent input and output messages between blocks. The focus of BPEL is on the control flow only, as the management of data is demanded to single executed services.

Event-driven process chains (EPCs) [131] is an informal graph-based notation for process specification widely used in industry. Nodes are distinguished into activ-

ities, avents and connectors, while edges represent the control flow. EPC schemas are not directly executable and since they provide a limited set of control flow constructs, industry demanded for extended EPCs (eEPCs) that are augmented with capabilities to model data objects, organizational entities and interfaces to processes.

eEPC models data objects as an optional graph node that does not belong to the control flow, and hence no support for the data lifecycle is provided. Niether EPC nor eEPC have a precise execution semantics and the semantics of data objects in eEPC is not formalized either.

The motivation for the development of *Yet Another Workflow Language* (YAWL) [127] was the lack of a process language that directly supported all control flow patterns identified by the workflow patterns initiative. Its main feature is having a formal syntax (defined through the use of set theory and predicate logic) and a formal semantics, defined in terms of a colored Petri net that interprets the YAWL specification.

Given that YAWL has been developed focusing on the control flow, data is considered only at the execution level and managed by the YAWL engine as set of process variables that can guide process' decisions. Hence the role of data is not primary.

1.1.2 Data-centric process models

The consciousness of lack of data supports stimulated the BPM community to study and develop on integrating processes and data. In this Section, starting from the analysis in [117, 89] we briefly survey the main efforts that has been recently done in this direction.

A *proclet* [132] is an object-specific process modeled as a Petri net, and can be seen as a workflow equipped with a knowledge base containing information on interactions with other proclets. Interactions between proclets are based on messages exchanged through ports and each message sent or received is stored in the knowledge base of the respective proclet.

More in detail, a proclet class describes the life-cycle of its proclet instances and is modeled in term of workflow nets with an explicit representation of interaction between proclets. Transitions/tasks can send and receive messages via ports while tasks can query the knowledge base and they have assigned pre- and post-conditions referring to the knowledge base. The execution semantics is given in terms of workflow nets and, hence, Petri nets.

Even though proclets have an essentially activity-centric nature, the use of pre- and post-conditions for activities, based on the information for knowledge base enables a sort of data-driven coordination of proclet instances.

The recently introduced *PHILarmonicFlows* [88] aims at integrating all entities involved in a data-aware business process system: data, processes, functions and users. PHILharmonicFlows use a relational data model, based on object types, their attributes and relation types to represent data. To describe the dynamics of the system, on the one hand to each object type a so-called micro-process must be specified, that defines the behavior of the corresponding object instances. A

micro-process is modeled as a finite-state machine, where states comprises several micro-steps that write attributes of the specific object instance. Transitions between states are triggered according to values of the attributes. On the other hand, macro-processes coordinates the interactions among objects instances. Again, they are defined by means of a finite-state machine where states refers to object instances and transitions represents interactions between them.

From a user perspective, users are provided with both data- and process-oriented views, i.e., they can access data at any point in time and they may invoke activities needed to proceed with the execution of micro process instances.

The *business artifact* framework [109, 81, 23, 44] provides a process design methodology that focuses on business relevant entities, called *artifacts*, rather than on activities. An artifact is characterized by the *information model* and the *lifecycle model*[2]. The information model holds all the business relevant information of interest ad it can be modified by *tasks*. The *lifecycle model* contraints the application of tasks, end hence, it specifies the possible way the artifact can evolve over time.

Artifact instances evolve in an environment that includes users, external data sources, services and events. Moreover, *relationships* capture links between artifacts and events.

The general framework for artifact-centric systems does not restrict the way to specify the information models and lifecycles. However, while the information model is usually a set of (possibly nested) attributes, the lifecycle can be specified in several ways. The work in [44, 23] illustrates a procedural state machine-based approach, however a declarative one, called GSM, will be analyzed in detail in Chapter 4.

The business artifact framework is a data-aware framework given that data are explicitly and mandatorily represented in the model, they evolve according to a lifecycle and relationship between instances are captured. In Chapter 6 we will see how it is possible to express also meta-level relationships between data in an artifact-centric system by means of an ontology.

1.2 Formal analysis and verification of artifact-centric process models

This thesis explores formal analysis of artifact-centric business process models. Our analysis amounts to checking *temporal* properties (as they refer to evolutions of the system over time) on a schema of a business process. This problem is usually referred to as *verification*.

The analysis can be performed *offline*, meaning before the real execution of the schema, and hence it takes into account all possible evolutions of the system, or at *runtime*, i.e., by observing a specific execution only. Technically, since in the first case all executions of a system are examined to check the satisfaction of a given property, the theoretical underlying problem is language inclusion, while in

[2]The authors use terms *meta-model* and *model* for which we refer to as *model* and *schema*. In this thesis, even though it may generate confusion, we stick to the original terminology and, only in the context of artifact-centric systems, we will use the terms *information model* and *lifecycle model* instead of *information schema* and *lifecycle schema*.

the second case, given that only the current execution is analyzed, the problem of interest is word recognition.

Model checking has been a major breakthrough in (offline) verification [20, 43, 12]. It addresses the problem of checking automatically a property on a schema of a system. Most model checking techniques require the dynamic system to verify to be finite-state, since they verify properties by systematically exhaustive exploration of the mathematical schema that describes the system. Typically symbolic techniques are used to reduce the cost of the state space exploration [28]. Often, even if finite, the system state spaces are in practice too large, and require the use of smart abstraction techniques, such as symmetry abstraction, abstract interpretation, symbolic simulation and abstraction refinement, to make such analysis very effective in practice [41].

Verification of artifact-centric process models is challenging for two main reasons. On the one hand the verification language should be expressive enough to both query complex data structures, such as a relational database, and should comprise temporal operators to assert constraints on the possible evolutions. On the other hand the presence of possibly unbounded data makes the usual analysis based on model checking of finite-state systems impossible in general, since, when data evolution is taken into account, the whole system becomes infinite-state.

Some verification techniques, including model checking, deal also with infinite-state systems and a number of solutions have been proposed to deal with state infiniteness. Many of them are based on identifying interesting classes of *transition systems* [43], definable by suitable formalisms, and some respective classes of decidable properties. For example, *well-structured transition systems* [1, 66] is an infinite-state mathematical structure where decidability of verification is guaranteed for a restricted class of properties only. Other results obtain decidability by suitable manipulation of basic transition systems, such as transductions [45], tree-iteration [137] and unfoldings, that can be intuitively thought as operations that build transition systems out of transition systems by preserving some regularities which can be exploited by verification algorithms. Another approach is regular model checking [25, 85, 24], a uniform paradigm for algorithmic verification of several classes of parameterized and infinite-state systems.

Unfortunately, mixing a rich query capability on an artifact state with the evolution given by the its lifecycle makes artifacts infinite-state systems of a different nature with respect to the ones mentioned above. Works on artifact-centric, or, more in general, data centric systems verification have been sparse, since they require knowledge of both dynamic systems and databases, but the issue is increasingly attracting interest lately.

One of the first work in this area is [3], where business models are specified as a so-called *relational transducer* that are state machines where states are relational databases and transitions map input sequences of relations (representing the interaction with the outside world) to output sequences of relations. The kind of analyses the authors carry out over transducers are checking temporal properties, expressed in a sort of linear-time temporal logic (LTL) [115, 136] with past operators, and log validity, which amounts to checking whether a given sequence of output relations can actually be generated by some input sequences. Such properties are decidable

for the semi-positive outputs and cumulative states (*SPOCUS*) transducer class, that, technically, generates regular languages that are accepted by finite automaton with self-loops only.

Work on formal analysis of restricted forms of artifact-centric systems has also been reported in [23, 35, 72, 73]. In [23] the authors check whether an artifact modified by services *successful completes* or if there are *dead-end paths*, and decidability is obtained posing restrictions on services, such as trivial, i.e., true, or not negative preconditions. In [35] the problem of checking whether one artifact-centric workflow may emulate the possible behaviors of another one is shown, and decidability is guaranteed if the infinite domain of artifacts' attributes is ordered.

Verification of more general properties, expressed in a computation tree logic (CTL) [40] like language are analyzed in [72, 73]. Unfortunately, decidability for the full language is obtained by bounding the domain, and for unbounded (but yet ordered) domains restricting pre- and post-conditions to be propositional (hence not referring to data values) is required. All of these frameworks do not model an underlying database.

The work reported in [57] is the most closely related to ours. Data in artifacts is taken into account (the infinite domain is equipped with a dense linear order) and it is essentially stored in two kinds of relations: attribute relations and state relations. Services make artifacts evolves, they are expressed as pre- and post-conditions and can write new data into state relations. The language used for verification purposes is a first order extension of LTL. In such a powerful framework, decidability is obtained by imposing restrictions on pre- and post-conditions of services: status relations (where new data may accumulate) can be queried only by checking if a given tuple of constants is contained.

In [46] the authors add data dependencies and arithmetics to the framework presented in [57], that are accepted to be important in real-life scenarios. Unfortunately with this extension, artifact systems can simulate counter machines, which variables on registers can be incremented unboundedly. Intuitively, to obtain decidability, it is necessary to break the mechanism which allows to update artifact variables depending on their own history: the so-called feedback-free processes do so by requiring updates of a variable to depend on the values of other variables only.

1.3 Reasoning about actions

Analysis of dynamic systems has been extensively studied in artificial intelligence, given that a formalism being able to capture not only a complex description of the world, but also how it is expected to change by means of actions, was required in the field of cognitive robotics. A representation of the world is the knowledge base of an agent which, by performing some action on the world it lives in, is expected to change the world in order to reach various objectives. The affinity of this field to ours is straightforward and notably the results presented in this thesis not only impact business process modeling (and more in general, processes on data bases) but also reasoning about actions, as briefly as briefly discussed in Section 3.6.

The most acclaimed and established logical approach in reasoning about actions is rooted in work of McCarthy [105], where a dialect of first-order logic, intended

to model actions, called *situation calculus*, is proposed. The main purpose of such a formalism was to provide a solution for the planning problem. The core notions of situation calculus reported here are taken from [118, 26].

Situation calculus has basically three distinguished kinds of first-order terms: *actions*, *situations* and *fluents*. Actions are functions that change the world and can be parametrized. Situations represent a possible world history, and are denoted by the special constant S_0 denoting the initial situation, and a function symbol $do(a, s)$ which denotes the new situation resulting from performing action a in situation s. Fluents are predicates or functions whose values change depending on situations. They are denoted by predicate and function symbols taking a situation term as their last argument.

The goal of *situation calculus* is ambitious: the axiomatization of a theory of actions which reflects as much as possible the real world. Among all the challenges, three main problems have been identified: *the qualification problem*, *the ramification problem* and the *frame problem*. The qualification problem arises when specifying preconditions of actions. Which are the conditions to be true in order for an action to be possible? One should take into account that the engines of the robot work, that the block is not glued to the floor, that an earthquake is not happening, and much more. An exhaustive listing of all qualification is practically infeasible and such an issue has no straightforward way to be addressed. The ramification problem is somehow the symmetric of the qualification but it focuses on the effects of an action, rather than on the preconditions. What are the consequences of an actions? If a robot enters into a room, not only it is currently in the room, but also the objects it is carrying are in the room, and its engine is in the room as well and it can increase the temperature of the room, and the robot's wheels can dirty the floor, and so on. Lastly, the ramification problem is strictly related to the frame problem, which addresses the issue of how to specify all the fluents that are unaffected by actions. For example, if a robot pickups a block, the block does not change color, the other blocks remains in the same position, the door stays closed, and so on. A trivial solution would amount to enumerate all fluents that are unaffected by each action effect, leading to a number of axioms that is roughly the number of actions times the number of fluents. Reiter [118] proposes an elegant and compact solution to these problems. Concerning the qualification problem, only the "important" qualification are mentioned, while the minor ones are dropped. A similar strategy is applied also to the ramification problem. The solution to the frame problem is more involved, and consists in writing a *successor state axiom* for each fluent F, which characterizes the value of F in the successor state resulting from performing a parametric action a in situation s.

Putting all together, a possible axiomatization for a theory of actions has one precondition axiom for each action, one successor state axiom for each fluent and unique name axioms for actions, which states that each action has a different name. With such an axiomatization at hand, it is possible to synthesize a solution for a planning problem as a side effect of theorem proving: it is enough to ask whether the axioms logically entails a situation s in which the goal is true to obtain a witness of s that is a plan for the goal.

Even though situation calculus provides a theoretical characterization of planning, it is not normally used to solve practical planning problems. Planning in the

AI community is generally done using variants of a formalism called STRIPS, from STanford Research Institute Problem Solver. STRIPS is dated 1960s, and many different varieties has been proposed over the years. However, all of those agree on some core aspects: *(i)* histories are not represented, but just the current snapshot of the world is, called (STRIPS) *database*; *(ii) operators* represent actions which, when applied to the current snapshot, produce a new database that describes the world after the action is performed. Such a formalism is more direct than situation calculus, because it provides a *progression* mechanism for a database. In STRIPS, the database is a set of ground atomic formulas (or facts) other than the equality predicate which can be thought as ground fluents with the situations argument suppressed. Hence differences between fluent and predicates disappear. Moreover, actions are not explicitly represented into the model, and hence it is not possible to directly reason about them. STRIPS operators are specified by preconditions and postconditions. Preconditions are sets of atomic formulas that need to hold before the action can apply, while postconditions are composed by a *delete list* which is a set of atomic formulas to be removed from the database and a *add list*, a set of formulas to be added to the database. Action parameters can appear as variables in the add and delete list, and hence they are not logical, but rather schema variables. An operator description is indeed a schema, meaning that it stands for all possible instances formed by instantiated its schema variables by constants in the domain. Schema variables are the only kind of variables allowed in operator descriptions atoms. The definition of the predicates and operators constitute the so-called *planning domain*, while the *planning problem* defines the initial state, the goal and the set of possible actions.

1.4 Contributions and thesis structure

Recalling the process lifecycle in the previous section, in the first part of the thesis we focus on the first phase of the process lifecycle, the *design and analysis* phase, by studying offline verification of artifact-centric systems. In the last chapter we address problems related to the third phase, the *enactment and monitoring* phase, by presenting a flexible technique for runtime monitoring of processes.

More precisely, in the first part we propose a novel framework for modeling artifact-centric systems that comprises three main elements for an artifact: *(i)* the *information model*, that is a structure where data is stored, *(ii) actions* (or *tasks*) that describe how such data can be modified and *(iii)* the *process* that constraints the application of actions and hence characterizes, in a declarative way, all possible evolutions of the artifact. Besides, we define a powerful verification language that is a first-order variant of μ-calculus [62, 125].

We explore different variations of the aforementioned setting through the thesis, that mainly differ from each other in the way data are represented or in the way actions are specified. Such differences lead to slightly different semantics for actions and processes and different constraints for achieving decidability of verification.

The last chapter turn to runtime verification and initiate the study of runtime monitoring of data evolving over time.

- In Chapter 2 we study a family of artifact-centric services whose task specifi-

cation are based on the notion of *conjunctive queries*.

In such a framework, the information model is a full-fledged relational database, and the lifecycle is specified in a declarative way as a set of condition-action rules. Conditions are evaluated on the current *snapshot* of the artifact, i.e., the current state of the database. Actions are *task* invocations, that query the current snapshot and generate the next one, possibly introducing new existential values representing inputs from the outside world. Since such values are yet unknown at analysis time, they are represented as nulls. The behavior of tasks is characterized using *conjunctive* pre-conditions and effects (or post-conditions), hence negation is not allowed.

On top of such a framework, we introduce a powerful verification logic based on a first-order variant of μ-calculus to express temporal properties. μ-calculus is well known to be more expressive than virtually all temporal logics used in verification, including CTL, LTL, CTL* [63], propositional dynamic logic (PDL) [67], and many others. For this reason, our results for μ-calculus immediately carry over to all these other logics, giving us a very robust approach.

Research in data exchange and data integration [65, 92] has deeply investigated the mapping between databases expressed through correspondences between conjunctive queries, consisting of the so-called tuple-generating dependencies (tgds) in the database jargon [2]. The core idea here is to consider the current state of data, and their state after the performance of a task, as two databases related through a set of the so-called *tuple generating dependencies* (tgds), a well-known notion in data exchange and data integration [65]. In this way we can exploit a syntactic condition called *weakly-acyclicity* of tgds [65, 58, 106, 103, 104] to get decidability of verification.

- The framework presented in Chapter 3 aims at overcoming the limitation of the previous one by adding negation in preconditions and effects of actions.

  The information model is again a relational structure, but the choice of allowing full first-order queries requires us to abandon the theory of conjunctive queries and homomorphisms that is at the base of the results presented in the previous chapter. The values introduced by actions are no longer modeled as null values but they are now Skolem functions, we thus assume actions to be deterministic, i.e., they generates equal values if fired on the same instance. The verification formalism is again a variant of μ-calculus where quantification ranges over elements contained in the initial database instance only. Decidability of such a language is, again, obtained by requiring a syntactic condition on actions analogous to weakly-aciclicity in data exchange.

  We then present a variant of the relational setting called *Relational Data-Centric Dynamic Systems* (DCDSs), studied in [11], that takes the core ideas of the frameworks presented in Chapters 2 and 3 and considers both the case in which actions behave deterministically and the case in which they behave nondeterministically. Syntactic restrictions guaranteeing decidability of verification are shown for both cases.

We now concretize this formal framework by reducing well-known BPM models

expressing data to DCDS.

- Chapter 4 presents in detail a concrete artifact model called Guard-Stage-Milestone (GSM) [82, 47]. The main feature of GSM is to incorporate intutive business-level constructs directly in model, in order to be easy to use, but still provide an operational semantics that ensure unambiguity.

  The information model in GSM is a set of attributes and the lifecycle is expressed in a declarative way, using constructs called *milestones*, *stages* and *guards*, that have been recently adapted by the Object Management Group as the standard for the Case Management Model Notation (CMMN) [112]. The operational semantics is formalized by specifying how the consumption of an event affects the information model of the system during a so-called business step (B-step).

  We then show how a reduction from GSM schemas to DCDS schemas. This is important for two reasons. In a first place it proves that the foundations of our framework, even though quite theoretical, are capable of capturing concrete artifact-centric models. Besides, it gives a procedure to analyze GSM schemas: verification of GSM schemas is, in general, undecidable, but once traduced in a DCDS it is possible to exploit the results in [11] for decidability of verification. Actually, we present a syntactic condition that is checkable directly on a GSM schema and that, being subsumed by the conditions for DCDSs, guarantee decidability of verification.

- Chapter 5 introduces Colored Petri Nets (CPNs) [84], a formalism that is born in response to the need of data in BPM. CPNs shows an effort in adapting a preexisting activity-centric formalism, namely Petri nets, in order to include data. In a CPN, tokens are colored, meaning that they carry data of a specific type and hence conditions on transitions can now consider not only the number of token in a specific place but also their data.

  Verification of CPN is undecidable in general, and we show a set of customary behavioral properties of CPN and some techniques used to check them. Unfortunately, decidability comes only when strong restrictions on data are imposed, such as bound on number of token and colors.

  We then informally discuss, analogously to Chapter 4, how to capture CPNs with DCDSs. Expressing a CPN as a DCDS is of extremely interest because it allows to verify very expressive properties. Indeed, on the one side the verification language in [11] subsumes the behavioral properties usually checked on CPNS, and goes beyond, and on the other side restrictions to achieve decidability for DCDSs require a bounded number of values in a state (or marking) but they do allow executions in which infinitely many distinct values occur in different states.

Next we enhance artifact-centric systems by introducing a semantic component.

- Chapter 6 follows the line of Chapters 2 and 3 by proposing a framework where artifacts are enriched with a semantic layer.

The information model of our semantic artifacts is a full-fledged ontology specified in a Description Logics (DL) [6] that is at the core of the Web Ontology Language OWL 2 QL. The ontology is composed by a TBox, that captures intentional information on the domain of interest, and by an ABox that acts as a storage. Data in the ABox are accessed by relying on query answering through ontologies, that, being able to deal with incomplete information, is based on logical implication (certain answers). As a query language, we use unions of conjunctive queries, possibly composing their certain answers through full FOL constructs. This gives rise to an *epistemic query language* that asks about what is "known" by the current knowledge base (KB) [30], i.e., the ABox. The KAB then contains *actions* whose execution changes the state of the ABox. Actions are specified as sets of conditional effects, where conditions are (epistemic) queries over the KB and effects are expressed in terms of new ABoxes. Action effects may contain special *Skolem terms* that represent new unknown individuals. A process, composed by a set of condition/action rules is once more used to specify which actions can be executed at each step.

A strong difference between this framework and the ones presented Chapters 2 and 3 lies in the treatment of equality. Indeed, equality constraints can now be expressed in the TBox and hence, as the system progresses and new information is acquired, distinct Skolem terms may be inferred to denote the same object.

The verification language is a first-order variant of μ-calculus, where atomic formulas are epistemic queries over the ontology that can refer both to known and unknown individuals, and where a controlled form of quantification across states is allowed. Verification of such a language is, in general, undecidable but it can be reduced to finite-state model checking when actions satisfy a form of weakly-aciclicty.

We then move to the problem of runtime verification of data-aware processes.

- Chapter 7 presents an initial study that aims at monitoring properties of business process data at runtime.

  More precisely, given a relational data schema, a property over the schema, and a process that modifies the data instances, we address the problem of enforcing the property during each step of the runtime execution of the process. In our setting, property are expressed using first-order linear temporal logic (FO-LTL) and they are evaluated under finite and fixed domain assumptions.

  Runtime verification of propositional temporal properties have been recently studied: in [14] an automata-based technique to solve such a problem is shown, and amounts to build a so-called *monitor* that is based on the Büchi automata of the formula [136]. In principle, since we consider bounded data, it is possible to naively monitor the first-order property by enumerating all possible assignment for the variables, i.e., by first "propositionalizing" the first order formula and then build the monitor. Unfortunately this would require an exponential blow-up in the size of the domain.

Instead, we propose an automata-based technique that is exponential in the size of the formula only. Technically, we build a compact automaton that, along with some auxiliary data structure, evolves together with data evolution and is able to monitor the property.

Chapters have a uniform structure throughout the thesis: a brief introduction initiates the reader to the dissertation, then the framework (or model) is presented, followed by technical results and a final discussion closes the chapters by emphasizing differences with other approaches and considering related work.

Part of this thesis has already been published in [36, 8, 83, 9, 80, 51, 10, 124, 56].

We also studied automated service composition without data. Automated service composition consists in automatically synthesizing an *orchestrator* that instructs, in an interleaved fashion, services to perform actions in order to simulate a target behavior. In [55] we applied the so-called "roman model" framework [75, 22, 114] to a practical scenario where services are embedded devices that collaborate with human users in a person-centric smart environment. In [50] we started from the "roman model" framework where services are modeled as a finite-state machine and we dropped the assumption of full observability of services states. We give a sound and complete procedure to synthesize the orchestrator in such a case, and we characterize the computational complexity of the problem. The procedure is based on working with belief (or knowledge) states, that is a standard technique used in artificial intelligence to tackle with incomplete information.

Falling outside the topic of data-aware business processes, such works has not been included.

Chapter 2

Conjunctive artifact-centric processes

Taking the core concepts from recent proposals by Hull et al. [23, 69, 35], we consider an artifact as formed by a *data component*, which describes its static part, and by a *lifecycle*, which characterizes the dynamic aspects. In our framework, the data component is a full-fledged relational database, and the lifecycle is specified, in a declarative way, as a set of condition-action rules. Conditions are evaluated on the current *snapshot* of the artifact, i.e., the current state of the database. Actions are *task* invocations, that query the current snapshot and generate the next one, possibly introducing new existential values representing inputs from the outside world. Since such values are yet unknown at analysis time, they are represented as nulls. Similar to the context of semantic web services [123], and deeply rooted in the literature on Reasoning about Actions in AI [118], here the behavior of tasks is characterized using pre-conditions and effects (or post-conditions).

The key point of our proposal is that both pre-conditions and effects are expressed as conjunctive queries. This implies that: *(i)* we query only positive information in the current state of the artifact (negation is not allowed in conjunctive queries), and *(ii)* we do not get disjunctive information as the effect of executing a task, though we get existential values, i.e., nulls, as the result of introducing unknown input from outside. The latter, *(ii)*, assures that the state generated by the execution of a task is still a relational database, even if it contains nulls. Such an assumption can often be made, and, in particular, in all those applications in which we have (almost) complete information on the result of executing a task. Instead, the former, *(i)*, might be a restriction in practice: it limits the way we can formulate queries on the current state and hence the way we can specify tasks. Chapter 3 will move in this direction by extending the approach presented here to full first-order queries in pre-conditions of tasks. However, this extension requires a much more sophisticated technical development that hides the core technique we show in this chapter. Focusing on conjunctive queries allows for exposing, in the cleanest and most elegant way, the very idea at the base of the approach, that is, the correspondence between tasks execution and data exchange and data integration [65, 92]. In both such fields, even though for slightly different purposes, there is the need for expressing relations between data elements of a schema, called *global* or *source schema*

and data elements of another (different) schema, called *local* or *target schema*. Such relations are captured by means of *assertions* that can be of three different kinds: *local-as-view* (LAV), *global-as-view* (GAV) or *global-and-local-as-view* (GLAV). A LAV assertion relates one element of the local schema to a query, i.e., a view, over the global schema, a GAV assertion relates one element of the global schema to a query over the local schema, and a GLAV assertion, generalizing the previous two, relates a query over the local schema to a query over the global schema. A data exchange setting is characterized by a source and a target schema and a set of source to target dependencies, that are GLAV assertions, called tuple-generating dependencies (tgds). The data exchange problem consists in, given an instance of the target schema and a set of dependencies, finding an instance of the target schema such that all the dependencies are satisfied. The dependencies are actually used by a procedure called *chase* to build a solution by repeatedly applying them until a specific condition is met.

Getting inspired by the use of tgds for building a new schema instance, the core idea of our work is to consider the current state of data, and their state after the performance of a task, as two databases related through a set of tgds. This view allows us to leverage on conditions that guarantee *finite chase of tgds* [65, 58, 106, 103, 104] to get decidability results even for very powerful verification languages.

On top of such a framework, we introduce a powerful verification logic based on a first-order variant of μ-calculus [100, 113, 62, 27] to express temporal properties. μ-calculus is well known to be more expressive than virtually all temporal logics used in verification, including CTL, LTL, CTL*, PDL, and many others. For this reason, our results for μ-calculus immediately carry over to all these other logics, giving us a very robust approach.

This setting, while quite expressive and inherently infinite-state, admits decidable verification under a reasonable restriction on the form of the effects of tasks, called *weak acyclicity* [65]. The crux of the result is that conjunctive queries are unable to distinguish between homomorphic equivalent databases: this can be exploited to bound the number of distinguishable artifact states. Thus, we can reduce verification to model checking of a finite-state transition system, which acts as a faithful abstraction of the original artifact.

2.1 The framework

Conjunctive artifact-centric services are services based on the notion of artifact, which merges data and processes in a single unit. An artifact is a tuple $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ whose components are:

- The *data component* $\mathcal{D}$, which captures the information of the artifact;
- the set of *tasks* $\mathcal{T}$, which manipulate artifact data;
- the *lifecycle* $\mathcal{C}$, which specifies the actual process of the artifact in terms of tasks that can be executed at each state.

Next, we define each artifact component in detail.

Artifact data component. An artifact data component $\mathcal{D} = \langle \mathcal{S}, I_0 \rangle$ is formed by a data schema $\mathcal{S}$ and an instance I_0 conforming to $\mathcal{S}$. The *data schema* has a relational structure and consists in a finite set of *relational (predicate) symbols* $R_1, \ldots, R_n$ (each one with an associated arity) and a finite or countably infinite set of *constant symbols* $\vec{c} = c_1, c_2, \ldots$.

A *data instance*, or simply *instance* (including the initial one I_0) over $\mathcal{S}$ is a standard first-order interpretation with a *fixed interpretation domain*. More precisely, a data instance is a pair $I = \langle \Delta, \cdot^I \rangle$ where:

- Δ is a countably infinite *domain*, fixed a-priori and shared by every data instance and it is partitioned into two countably infinite disjoint sets $const(\Delta)$ of *constants* and $ln(\Delta)$ of *labeled nulls*. The first set is used to interpret constant symbols, while the second one is for interpreting existentials (see later);
- $\cdot^I$ is an interpretation function that associates:
  - to each constant symbol c, a constant $c^I \in const(\Delta)$ such that for each $c_1, c_2 \in const(\Delta)$ if $c_1 \neq c_2$ then $c_1^I \neq c_2^I$, namely we adopt the *Unique Name Assumption*. Furthermore, we require the interpretation of constants to be rigid, that is, given I and I', we have that $c_n^I = c_n^{I'}$ for each constant symbol c_n. Thus, we blur the distinction between constant symbols and constants in $const(\Delta)$;
  - to each m-ary relation symbol R_i a finite m-ary relation ${R_i}^I \subseteq \Delta^m$.

An artifact data instance is a relational database, as the function $\cdot^I$ lists all tuples belonging to each relation. An expression $R_i(d_1, \ldots, d_m)$ is called a *fact*. We say that a fact belongs to an interpretation I iff $\vec{d} = \langle d_1, \ldots, d_m \rangle \in R_i^I$, so to characterize interpretation functions from the set of their facts (notice that such sets are finite). Following the database literature [2], we call *active domain* $\bar{\Delta}_I$ of an instance I the set of domain elements appearing in the facts of I.

Example 2.1.1. *We model banks that provide services to their customers, such as loans or money transfers. Each service has a cost that has to be paid in advance. When a customer inquires for the provision of a service, it first has to be approved by a supervisor, then it is paid by the customer, and finally it is provisioned by the bank. Special "premier customers" do not need the supervisor approval. The artifact schema $\mathcal{S}$ consists of the following relation symbols:*

- Customer($\underline{custSsn}, name$)*, which contains customers information;*
- Service($\underline{servCode}, cost$)*, which contains information about the different types of services that the bank offers to its customers;*
- ServiceClaimed($\underline{servCode}, custSsn$)*, which keeps track of information of services requested by clients;*
- Examined($\underline{servCode}, spvName, outcome$)*, which contains the names of supervisors in charge of evaluating customers' claims;*
- Payment($\underline{servCode}, custSsn, amount$)*, which contains information about service payments;*

- ServiceProvided($\underline{servCode}$, *custSsn*), *which holds the services which have been provided;*
- PremierMember($\underline{custSsn}$), *which contains "premier" customers;*
- Account($\underline{accId}$, *custSsn*, *maxWithdrawal*, *creditCard*), *which holds information about bank accounts.*

Initial instance I_0 *is an interpretation of the above relations:*

- $\mathsf{Customer}^{I_0} = \{\langle 337505, JohnSmith\rangle, \langle 125232, MaryStewart\rangle\}$, *and*
- $\mathsf{Service}^{I_0} = \{\langle L057, 100\rangle, \langle L113, 150\rangle, \langle C002, 50\rangle\}$,

and all other relations are empty. ■

In order to query data instances, we use *conjunctive queries*, a special and widely used class of first-order formulas corresponding to relational algebra select-project-join queries. A *conjunctive query* is a formula *cq* of the form:

$$\exists \vec{y}.body(\vec{y}, \vec{x})$$

where *body* is a conjunction of atoms (atomic formulas) involving constant symbols, existentially quantified variables $\vec{y}$ and free variables $\vec{x}$. A conjunctive query returns, as answer, the domain elements (both constants and nulls) that, when substituted to the free variables, make the formula true in the current instance. More formally, given an artifact instance $I = \langle \Delta, \cdot^I \rangle$, the *answer to a conjunctive query* $cq(\vec{x})$ with free variables $\vec{x}$, over I, denoted by $cq(\vec{x})^I$ is defined as:

$$cq(\vec{x})^I = \{\eta \mid \langle I, \eta\rangle \models cq(\vec{x})\}$$

where $\eta : \vec{x} \to \Delta$ is an assignment to the free variables.

The notion of homomorphism [37] plays a key role in our setting, so we remind its definition here. Given two instances $I_1 = \langle \Delta, \cdot^{I_1} \rangle$ and $I_2 = \langle \Delta, \cdot^{I_2} \rangle$ over the same schema $\mathcal{S}$, a *homomorphism* from I_1 to I_2, denoted by $h : I_1 \to I_2$, is a function from Δ to Δ such that:

1. for every constant $c \in const(\Delta)$, we have that $h(c) = c$;
2. for every $\langle d_1, \ldots, d_m\rangle \in R_i^{I_1}$, we have that $\langle h(d_1), \ldots, h(d_m)\rangle \in R_i^{I_2}$.

Two instances I_1 and I_2 are *homomorphically equivalent*, written $I_1 \overset{h}{=} I_2$, if there exist two homomorphisms $h_1 : I_1 \to I_2$ and $h_2 : I_2 \to I_1$.

Intuitively an homomorphism $h : I_1 \to I_2$ preserves the interpretation of constants $const(\Delta)$ but not of labeled nulls $ln(\Delta)$ in I_1, which are mapped in a non-injective way, either to constants or nulls, in I_2. In other words, a homomorphism interprets nulls of I_1 as existential values.

The characterizing property of conjunctive queries, from the semantical point of view, is that they are invariant under homomorphic equivalence [2]. That is, if two data instances I and I' are homomorphic, then each boolean (without free variables) conjunctive query *cq* produces exactly the same (boolean) answer: $cq(\vec{x})^I = cq(\vec{x})^{I'}$.

The existential interpretation of labeled nulls given by homomorphisms suggests a different way of answering conjunctive queries, that essentially sees the set of facts in the interpretation as a *theory* where all nulls are treated as existential variables. To make this notion precise, given an interpretation I we define the (*infinite*) set W_I of interpretations $I' = \langle \Delta, \cdot^{I'} \rangle$ over $\mathcal{S}$ such that there exists an homomorphism $h : I \to I'$. We define the *certain answers* of a conjunctive query cq as:

$$cert^I(cq) = \bigcap_{I' \in W_I} cq^{I'}$$

that is, the tuples of elements in I returned by the query in each interpretation I' such that there exists an homomorphism $h : I \to I'$. It is easy to see that such tuples can only be formed the constants in $const(\Delta)$ appearing in the active domain of I, since these are the only elements in the answers that are preserved by homomorphism. Intuitively, when using certain answers, the current I is seen as a representative of a set of instances (those in W_I) and hence, when querying I, we return the set of tuples that make the query true in all those instances.

In our framework, we assume that the user can pose arbitrary conjunctive queries to the current instance, but require them to be evaluated through certain answers. In this way, we become independent of the particular null values occurring in the data instance, since they are not returned as answers, though they can still be used as witnesses of existentially quantified variables. On the other hand, when we evolve the artifact by executing a task, we do consider null values in the current instance as legitimate elements to be propagated to the next state according to task effects.

Artifact tasks. A task modifies the artifact data by means of effects, that are formalized as *tuple generating dependencies* (tgds) [2, 65]. An *effect specification* ξ over a schema $\mathcal{S}$ is a formula of the form:

$$\exists \vec{y}.\phi(\vec{x}, \vec{y}, \vec{c}) \to \exists \vec{w}.\psi(\vec{x}, \vec{w}, \vec{d})$$

where ϕ and ψ are conjunctions of atoms over $\mathcal{S}$; $\vec{x}$, $\vec{y}$, $\vec{w}$ denote sets of variables and $\vec{c}, \vec{d}$ denote set of constants occurring in ϕ and ψ. We call the left-hand side of ξ the *premise*, and the right-hand side the *conclusion*. Notice that both the premise and the conclusion are *conjunctive queries*.

We now define the semantics of such effects by showing how they are used to progress the current instance. We first illustrate the effect of enacting a single effect specification, and then we turn to the enactment of a task. Let $I = \langle \Delta, \cdot^I \rangle$ be an instance over $\mathcal{S}$, and $\xi = \exists \vec{y}\ \phi(\vec{x}, \vec{y}, \vec{c}) \to \exists \vec{w}\ \psi(\vec{x}, \vec{w}, \vec{d})$ an effect specification. The result of *enacting effect specification* ξ on I, is the set of facts $\xi(I)$ defined as follows:

> Let $\vec{\eta} = (\exists \vec{y}\ \phi(\vec{x}, \vec{y}, \vec{c}))^I$, be the answer to the query $\exists \vec{y}\ \phi(\vec{x}, \vec{y}, \vec{c})$ in I, then for each $\eta_i \in \vec{\eta}$ we proceed as follows: for each atoms $R_i(\vec{x}, \vec{w}, \vec{d})$ occurring in ψ, we include in $\xi(I)$ a new fact $R_i^{I'}(\vec{x}, \vec{w}, \vec{d})|^{\psi}_{\eta_i}$, obtained by substituting every variable in $\vec{x}$ with the corresponding element given by the assignment η_i, and every variable in $\vec{w}$ with a fresh (not appearing elsewhere) labeled null $\ell \in ln(\Delta)$.

Intuitively, the premise acts as a query and selects domain elements, both constants and null values, from the active domain of the current instance, while the conclusion builds (a fragment of) the new instance by creating new facts using such elements. Notice that fresh labeled nulls are possibly introduced in the new facts as witnesses to existential values in the effect conclusion. Notice also that no contradiction can arise in generated instances, given that effects of tasks, being based on conjunctive queries, are only positive.

Next we define the semantics of task execution. A *task* T for a schema $\mathcal{S}$ is specified as a set $\vec{\xi} = \{\xi_1, \ldots, \xi_n\}$ of effect specifications. The result of *executing task* T on I, denoted by $I' = do(T, I)$, is a new instance $I' = \langle \Delta, \cdot^{I^T} \rangle$ on the same schema $\mathcal{S}$, obtained by the union of the enactments of each effect specification. Namely $I' = \langle \Delta, \cdot^{I'} \rangle$ where the interpretation function $\cdot^{I'}$ is characterized by the facts $\bigcup_{\xi \in \vec{\xi}} \xi(I)$.

Let us make some key observations on such tasks. First, the role of the existential quantification on the two sides of an effect specification is very different. The existential quantification on the premise is the usual one used in conjunctive queries, which projects out variables used only to make joins. Instead, the existential quantification on the conclusion is used as a witness of values that should be chosen by the user when executing the task. In other words, the choice function used to assign witnesses to the existential variables should be in the hands of the user. Here, since we do not have such a choice at hand, we introduce a fresh null, to which we assign an existential meaning through homomorphisms. The second observation is that we do not make any persistence (or frame [118]) assumption in our formalization. In principle, at every move, we substitute the whole old data instance with a new one. Nonetheless, it should be clear that we can easily write effect specifications that *copy* big chunks of the old instance into the new one. For instance, $R_i(\vec{x}) \rightarrow R_i(\vec{x})$ copies the whole extension of a relation R_i.

Example 2.1.2. *We continue our previous example by turning to the available tasks. As syntactic sugar, we include some input parameters (the symbols between parentheses after the task name). In order to execute a task, its parameters must be instantiated with constants as specified by the condition-action rules that form artifact lifecycle (see below). The tasks in our domain are the following:*

- ClaimService($custSsn$, $servCode$), *with effects:*

$$\{\ \exists x, y.\, \mathsf{Customer}(custSsn, x) \wedge \mathsf{Service}(servCode, y) \ \rightarrow \mathsf{ServiceClaimed}(servCode, custSsn),\ \mathsf{copyFrame}\}$$

  This task models the choice of the customer $custSsn$ to apply for the provision of a new service of type $servCode$. Since the resulting instance is a completely new one consisting of tuples specified by the task effects, we need to explicitly "copy" all facts that we do not require to be dropped after the task execution. This is done by effects of the form $R(x_1, \ldots, x_n) \rightarrow R(x_1, \ldots, x_n)$ for each relation $R \in \mathcal{S}$. We denote collectively such copying effects as copyFrame. *Intuitively, the result of firing task* Claim_service($cust\_ssn$, $serv\_code$) *on an instance I results in a new instance I' that not only contains I, but also*

includes the new tuple $\mathsf{ServiceClaimed}(custSsn, servCode)$ *provided that the premise is satisfied by* I*, otherwise* $I' = I$*.*

- $\mathsf{MakePayment}(custSsn,\ servCode,\ amount)$ *with effects:*

$$\begin{array}{l}\{\mathsf{ServiceClaimed}(servCode, custSsn)\ \rightarrow \\ \quad\ \mathsf{Payment}(servCode, custSsn, amount), \\ \mathsf{copyFrame}\}\end{array}$$

This task models the payment operation performed by a customer for a service that has been previously requested, i.e., it includes the tuple $\mathsf{Payment}(custSsn,\ servCode,\ amount)$ *in the resulting instance.*

- $\mathsf{GrantApproval}(servCode)$ *with effects:*

$$\begin{array}{l}\{\exists x.\mathsf{ServiceClaimed}(servCode, x)\ \rightarrow \\ \quad\ \exists z.\,\mathsf{Examined}(servCode, z, \text{“}approved\text{”}), \\ \mathsf{copyFrame}\}\end{array}$$

This task represents the approval of a service that has been requested, by including (according to its effects) the fact $\mathsf{Examined}(servCode, \ell, \text{“}approved\text{”})$ *where* ℓ *is a fresh labeled null that models a possible supervisor.*

- $\mathsf{ProvideServices}()$ *with effects:*

$$\begin{array}{l}\{\exists v, z.\mathsf{ServiceClaimed}(x, y) \wedge \mathsf{Examined}(x, v, \text{“}approved\text{”}) \wedge \\ \mathsf{Payment}(x, y, z) \wedge \mathsf{Service}(x, z)\ \rightarrow \mathsf{ServiceProvided}(x, y), \\ \mathsf{copyFrame}\}\end{array}$$

This task models the delivery of all services that have been explicitly approved by a supervisor and that have been already paid.

- $\mathsf{QuickService}()$ *with effects:*

$$\begin{array}{l}\{\exists z.\mathsf{ServiceClaimed}(x, y) \wedge \mathsf{Payment}(x, y, z) \wedge \mathsf{PremierMember}(y)\ \rightarrow \\ \quad\ \mathsf{ServiceProvided}(x, y), \\ \mathsf{copyFrame}\}\end{array}$$

This task delivers all the services for which the correct amount was paid and that have been requested by a premier customer.

- $\mathsf{AwardPremierStatus}()$ *with effects:*

$$\begin{array}{l}\{\exists y, t, u, w, z.\,\mathsf{Customer}(x, y) \wedge \mathsf{ServiceProvided}(z, x) \wedge \\ \quad\ \mathsf{Account}(u, x, w, t)\ \rightarrow\ \mathsf{PremierMember}(x), \\ \mathsf{copyFrame}\}\end{array}$$

This task awards the premier status to all customers holding a bank account who applied for the provision of a service that had already been accepted.

■

Artifact lifecycle. The artifact lifecyle is defined in terms of condition-action rules, that specify, for every instance, which tasks can be executed. A *(condition-action) rule* for a schema $\mathcal{S}$ is an expression ϱ of the form $\pi \mapsto T$ where π is a precondition and T is a task. The precondition is a *closed* formula over $\mathcal{S}$ defined according to the following syntax:

$$\pi ::= cq \mid \neg\pi \mid \pi_1 \wedge \pi_2$$

where cq is a boolean conjunctive query. Preconditions are arbitrary boolean combinations of boolean conjunctive queries interpreted under the certain answer semantics, namely we define the semantic relation *artifact instance I logically implies precondition π*, written $I \rhd \pi$, by induction on the structure of the precondition, as follows:

$$\begin{array}{lll} I \rhd cq & \text{iff} & cert^I(cq) = true \\ I \rhd \neg\pi & \text{iff} & I \not\rhd \pi \\ I \rhd \pi_1 \wedge \pi_2 & \text{iff} & I \rhd \pi_1 \text{ and } I \rhd \pi_2 \end{array}$$

Given a condition-action rule $\pi \mapsto T$ and an instance I, if I logically implies precondition π, then the task T is executable and, when executed, it generates a new instance I' according to T.

Observe that, while we disallow negation in task effects so as to exploit the theory of conjunctive queries, in the condition-action rules we do allow for it but we require the certain answers semantics. In this way we are only composing (using boolean connectives) the results of the conjunctive queries, and hence two homomorphic equivalent instances are guaranteed to satisfy the same conditions. Notice also that negation in this way becomes a sort of (stratified) "negation-as-failure" [39].

Example 2.1.3. *The* artifact lifecycle *of our running example is specified by the following condition-action rules:*

$\top(CustSsn, servCode) \mapsto \mathsf{ClaimService}(CustSsn, servCode)$
$\top(CustSsn, servCode, amount) \mapsto \mathsf{MakePayment}(CustSsn, servCode, amount)$
$\top(ServCode) \mapsto \mathsf{GrantApproval}(ServCode)$
$\exists x, y, v, w.\, \mathsf{Payment}(x, y, w) \wedge \mathsf{Service}(x, w) \wedge \mathsf{RequestExamined}(x, v, \text{“}approved\text{”}) \mapsto \mathsf{ProvideServices}()$
$\exists x, y, w.\, \mathsf{Payment}(x, y, w) \wedge \mathsf{Service}(x, w) \wedge \mathsf{PremierMember}(y) \mapsto \mathsf{QuickService}()$
$\exists x, y, u, w, t.\, \mathsf{ServiceProvided}(x, y) \wedge \mathsf{Account}(u, y, w, t) \mapsto \mathsf{AwardPremierStatus}()$

Again, we use parameters (occurring as free variables above) as syntactic sugar for a much larger set of condition-action rules obtained by instantiating the parameters to constants from a finite set. For example, such a set may contain all constants from the initial data instance of the artifact specified below, plus some extra ones used for convenience, e.g., to represent some predetermined amounts of money to be use for the parameter amount. ■

2.2 Conjunctive artifact execution

Let us consider an *artifact* $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$, with data component $\mathcal{D} = \langle \mathcal{S}, I_0 \rangle$ where $\mathcal{S}$ is the artifact data schema and I_0 is the initial artifact data instance. Moreover, let $\mathcal{I}$ be the set of all possible instances over $\mathcal{S}$.

Artifact execution tree. We can describe all possible executions of an artifact A by the so-called *execution tree* of A. The *execution tree* is a tuple $\mathfrak{T}_A = \langle \Sigma, \sigma_0, L, \Rightarrow \rangle$ where Σ is the set of states (or nodes), σ_0 is the root, $L : \Sigma \to \mathcal{I}$ labels states with data instances, and $\Rightarrow \subseteq \mathcal{I} \times \mathcal{T} \times \mathcal{I}$ is the transition relation that determines the successors of the current node. We use the notation $\sigma \overset{T}{\Longrightarrow} \sigma'$ for $\langle \sigma, T, \sigma' \rangle \in \Rightarrow$. The set Σ of states, its labeling L and the set $\Rightarrow$ of transitions are defined inductively as follows:

- the root is $\sigma_0 \in \Sigma$, with $L(\sigma_0) = I_0$;
- given a state σ for each task $T \in \mathcal{T}$ such that there exists a rule $\varrho = \pi \mapsto T$ such that $L(\sigma) \rhd \pi$, add state $\sigma'_T \in \Sigma$ with $L(\sigma'_T) = do(T, I)$ (i.e., $L(\sigma'_T)$ is the data instance obtained by applying T to $L(\sigma)$) and transition $\sigma \overset{T}{\Longrightarrow} \sigma'_T$.

Notice that such an execution tree is *infinite*, given that we can possibly keep executing tasks and, hence, producing new instances. Indeed, each state σ of the execution tree represent the whole history that generated it, namely the path from the root σ_0 to σ. Notice also that, when a new data instance $do(T, I)$ is generated from I, we are free to choose any fresh labeled null for the existential variables in the conclusion of the effect specifications. However, all such instances are *equivalent modulo nulls renaming*. Given two instances $I_1 = \langle \Delta, \cdot^{I_1} \rangle$ and $I_2 = \langle \Delta, \cdot^{I_2} \rangle$ over the same schema $\mathcal{S}$, a *nulls renaming* from I_1 to I_2, denoted by $r : I_1 \to I_2$ is an injective homomorphism, i.e., a function such that:

1. for every constant $c \in const(\Delta)$ we have that $r(c) = c$;
2. for every couple of different labeled nulls $\ell_1, \ell_2 \in ln(\Delta)$ we have that $r(\ell_1) \neq r(\ell_2)$ and
3. for every $\langle d_1, \ldots, d_m \rangle \in R_i^{I_1}$, we have that $\langle r(d_1), \ldots, r(d_m) \rangle \in R_i^{I_2}$

Two instances I_1 and I_2 are *equivalent modulo nulls renaming*, denoted by $I_1 \overset{mnr}{=} I_2$ iff they are isomorphic, i.e., iff there exists a nulls renaming $r : I_1 \to I_2$ such that its inverse r^{-1} is a null renaming from I_2 to I_1.

Given an artifact A, there exists a unique execution tree $\mathfrak{T}_A$ modulo null renaming.

Artifact transition systems and bisimulation. The execution tree is a special case of transition system. A *transition system* for A is a tuple $\mathfrak{A}_A = \langle \Sigma, \sigma_0, L, \Rightarrow \rangle$ where *(i)* Σ is the (possibly infinite) set of states; *(ii)* σ_0 is the initial state; *(iii)* $L : \Sigma \to \mathcal{I}$ is a labeling function that associates to each state in Σ a data instance in $\mathcal{I}$ and *(iv)* $\Rightarrow \subseteq \mathcal{I} \times \mathcal{T} \times \mathcal{I}$ is the transition relation.

Clearly not all transition systems for an artifact A represent the behavior of the execution tree $\mathfrak{A}_A$. To express which ones do, we use a suitable variant of *bisimulation* [108], tailored for our purposes: we indeed consider that the user can only query data instances through conjunctive queries, evaluated to return certain answers.

Given two transition systems for the same artifact A, $\mathfrak{A}_1 = \langle \Sigma_1, \sigma_{0,1}, L_1, \Rightarrow_1 \rangle$ and $\mathfrak{A}_2 = \langle \Sigma_2, \sigma_{0,2}, L_2, \Rightarrow_2 \rangle$, a *bisimulation* is a relation $B \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle \sigma_1, \sigma_2 \rangle \in B$ implies that:

1. for every conjunctive query cq we have that $cert^{L_1(\sigma_1)}(cq) = cert^{L_2(\sigma_2)}(cq)$;
2. if $\sigma_1 \stackrel{T}{\Longrightarrow} \sigma_1'$ then there exists σ_2' such that $\sigma_2 \stackrel{T}{\Longrightarrow} \sigma_2'$ and $\langle \sigma_1', \sigma_2' \rangle \in B$;
3. if $\sigma_2 \stackrel{T}{\Longrightarrow} \sigma_2'$ then there exists σ_1' such that $\sigma_1 \stackrel{T}{\Longrightarrow} \sigma_1'$ and $\langle \sigma_1', \sigma_2' \rangle \in B$.

We say that two states σ_1 and σ_2 are *bisimilar*, denoted as $\sigma_1 \sim \sigma_2$, if there exists a bisimulation B such that $\langle \sigma_1, \sigma_2 \rangle \in B$. Two transition systems $\mathfrak{A}_1 = \langle \Sigma_1, \sigma_{0,1}, L_1, \Rightarrow_1 \rangle$ and $\mathfrak{A}_2 = \langle \Sigma_2, \sigma_{0,2}, L_2, \Rightarrow_2 \rangle$ are *bisimilar* if $\sigma_{0,1} \sim \sigma_{0,2}$.

With the notion of bisimulation at hand, we can state that any transition system that is bisimilar to the execution tree represents the behavior of the artifact. We can exploit this fact to perform verification on a transition system that is more manageable than the execution tree. We will do so later: first, we introduce the verification formalism.

2.3 Verification formalism

To specify dynamic properties we use μ-calculus [62, 125], one of the most powerful temporal logics for which model checking has been investigated, and indeed is able to express both linear time logics, as LTL [115], and branching time logics such as CTL [40] or CTL* [63, 43]. The main characteristic of μ-calculus is the ability of expressing directly least and greatest fixpoints of (predicate-transformer) operators formed using formulas relating the current state to the next one. By using such fixpoint constructs, one can easily express sophisticated properties defined by induction or co-induction. This is the reason why virtually all logics used in verification can be considered as fragments of μ-calculus.

From a technical viewpoint, μ-calculus separates local properties, i.e., those asserted on current state or states that are immediate successors of the current one, from properties that talk about states that are arbitrarily far [27]. The latter are expressed through the use of fixpoints. Such a separation is very convenient for theoretical investigation, and indeed makes μ-calculus the language of choice for much theoretical work [62]. On the other hand, from a practitioner point of view, expressing properties using directly fixpoint can be cumbersome and, in most applications, simpler logics like CTL or LTL are preferred. For a thorough introduction to μ-calculus, we refer the reader to Stirling's book [125] which looks at μ-calculus both from the theoretical and from the practical point of view. The choice of using μ-calculus in our investigation allows for immediately transferring the results obtained to simpler logics like LTL, CTL, CTL*, etc.

We introduce a variant of μ-calculus, called $\mu\mathcal{L}$, that conforms to the basic assumption of our formalism: the use of conjunctive queries and certain answers to talk about data instances. This intuitive requirement can be made formal as follows: $\mu\mathcal{L}$ must be invariant with respect to the notion of bisimulation introduced above.

Given an artifact $A = \langle \mathcal{S}, \mathcal{T}, \mathcal{C} \rangle$, the *verification formulas* of $\mu\mathcal{L}$ for A have the following form:

$$\Phi ::= cq \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid [T]\Phi \mid \langle - \rangle T\Phi \mid \mu Z.\Phi \mid \nu Z.\Phi \mid Z$$

where cq is a boolean conjunctive query (interpreted through certain answers) over the artifact schema and Z is a predicate variable symbol.

The symbols μ and ν can be considered as quantifiers, and we make use of notions of scope, bound and free occurrences of variables, closed formulas, etc., referring to them. For formulas of the form $\mu Z.\Phi$ and $\nu Z.\Phi$, we require the *syntactic monotonicity* of Φ w.r.t. Z: every occurrence of the predicate variable Z in Φ must be within the scope of an even number of negation signs. In μ-calculus, given the requirement of syntactic monotonicity, the least fixpoint $\mu Z.\Phi$ and the greatest fixpoint $\nu Z.\Phi$ always exist.

In order to define the meaning of such formulas, we resort to interpretations that are transition systems. Let $\mathfrak{A} = \langle \Sigma, \sigma_0, L, \Rightarrow \rangle$ be a transition system for A with initial data instance I_0, and let $\mathcal{V}$ be a predicate valuation on $\mathfrak{A}$, i.e., a mapping from the predicate variables to subsets of the states in $\mathfrak{A}$. Then, we assign meaning to μ-calculus formulas by associating to $\mathfrak{A}$ and $\mathcal{V}$ an *extension function* $(\cdot)^{\mathfrak{A}}_{\mathcal{V}}$, which maps μ-calculus formulas to subsets of $\mathcal{I}$. The extension function $(\cdot)^{\mathfrak{A}}_{\mathcal{V}}$ is defined inductively as follows:

$$
\begin{array}{lcl}
(cq)^{\mathfrak{A}}_{\mathcal{V}} & = & \{\sigma \in \Sigma \mid cert^{L(\sigma)}(cq)\} \\
(Z)^{\mathfrak{A}}_{\mathcal{V}} & = & \mathcal{V}(Z) \subseteq \Sigma \\
(\neg\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = & \Sigma - (\Phi)^{\mathfrak{A}}_{\mathcal{V}} \\
(\Phi_1 \wedge \Phi_2)^{\mathfrak{A}}_{\mathcal{V}} & = & (\Phi_1)^{\mathfrak{A}}_{\mathcal{V}} \cap (\Phi_2)^{\mathfrak{A}}_{\mathcal{V}} \\
(\langle - \rangle T\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = & \{\sigma \in \Sigma \mid \exists \sigma'.\ \sigma \overset{T}{\Longrightarrow} \sigma' \text{ and } \sigma' \in (\Phi)^{\mathfrak{A}}_{\mathcal{V}}\} \\
([T]\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = & \{\sigma \in \Sigma \mid \forall \sigma'.\ \sigma \overset{T}{\Longrightarrow} \sigma' \text{ implies } \sigma' \in (\Phi)^{\mathfrak{A}}_{\mathcal{V}}\} \\
(\mu Z.\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = & \bigcap\{\mathcal{E} \subseteq \Sigma \mid \quad (\Phi)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]} \subseteq \mathcal{E}\ \} \\
(\nu Z.\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = & \bigcup\{\mathcal{E} \subseteq \Sigma \mid \quad \mathcal{E} \subseteq \ (\Phi)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]}\}
\end{array}
$$

Figure 2.1. Semantics of conjunctive $\mu\mathcal{L}$ formulas

Intuitively, $(\cdot)^{\mathfrak{A}}_{\mathcal{V}}$ assigns to the various constructs of μ-calculus the following meaning:

- The boolean connectives have the expected meaning.
- The extension of $\langle - \rangle T\Phi$ includes the states σ such that starting from σ, there is an execution of task T that leads to a successor state σ' included in the extension of Φ.
- The extension of $[T]\Phi$ includes the states σ such that starting from σ, each execution of task T leads to some successor state σ' included in the extension of Φ.
- The extension of $\mu Z.\Phi$ is the *smallest subset* $\mathcal{E}_\mu$ of Σ such that, assigning the extension $\mathcal{E}_\mu$ to Z, the resulting extension of Φ is contained in $\mathcal{E}_\mu$. That is, the extension of $\mu X.\Phi$ is the *least fixpoint* of the operator $\lambda\mathcal{E}.(\Phi)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]}$ (here $\mathcal{V}[Z/\mathcal{E}]$ denotes the predicate valuation obtained from $\mathcal{V}$ by forcing the valuation of Z to be $\mathcal{E}$).
- Similarly, the extension of $\nu Z.\Phi$ is the *greatest subset* $\mathcal{E}_\nu$ of Σ such that, assigning the extension $\mathcal{E}_\nu$ to Z, the resulting extension of Φ contains $\mathcal{E}_\nu$. That

is, the extension of $\nu Z.\Phi$ is the *greatest fixpoint* of the operator $\lambda\mathcal{E}.(\Phi)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]}$.

In expressing temporal properties using $\mu\mathcal{L}$ below, we use the following abbreviations: $\langle - \rangle - \Phi \doteq \bigvee_{T\in\mathcal{T}}\langle - \rangle T\Phi$ and $[-]\Phi \doteq \bigwedge_{T\in\mathcal{T}}[T]\Phi$, where $\mathcal{T}$ is the set of all tasks of the artifact. In this way, we can talk about *all next states* (resulting from every possible task execution) or about *some next states* resulting from certain task executions. With these abbreviations at hand, it is easy to express natural temporal properties such as "eventually a local property ϕ holds in all runs" (a *liveness* property):

$$\mu Z.\phi \vee [-]Z$$

or "always a local property ϕ holds (in all runs)" (a *safety* property):

$$\nu Z.\phi \wedge [-]Z$$

Notice that the negation of $\nu Z.\phi\wedge[-]Z$ is not $\mu Z.\phi\vee[-]Z$ but instead $\mu Z.\phi\vee\langle-\rangle-Z$ which expresses that eventually ϕ holds along some (but not necessarily all) runs.

Coming back to the first formula, $\mu Z.\phi\vee[-]Z$ can be seen as the "smallest solution" of the equation: $Z = \phi \vee [-]Z$ that is the smallest predicate that substituted to the variable Z makes the equation true. More formally $\mu Z.\phi \vee [-]Z$ denotes in every interpretation (i.e., transition system) $\mathfrak{A}$, the least fixpoint of the operator $\lambda\mathcal{E}.(\phi \vee [-]Z)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]}$, i.e., the smallest set $\mathcal{E}_\mu$ of states of $\mathfrak{A}$ that makes the equation $\mathcal{E}_\mu = (\phi \vee [-]Z)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}_\mu]}$ true. Similarly $\nu Z.\phi \wedge [-]Z$ can be seen as the "greatest solution" of the equation $Z = \phi \wedge [-]Z$, or more precisely, in every interpretation $\mathfrak{A}$ the greatest fixpoint of the operator $\lambda\mathcal{E}.(\phi \wedge [-]Z)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]}$, i.e., the greatest set $\mathcal{E}_\nu$ of states of $\mathfrak{A}$ that makes the equation $\mathcal{E}_\nu = (\phi \wedge [-]Z)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}_\nu]}$ true.

The reasoning problem we are interested in is *model checking*: *checking whether a $\mu\mathcal{L}$ closed formula Φ holds in an artifact A with initial data instance I_0.*

Formally, such a problem is defined as checking whether $L(\sigma_0) \equiv I_0 \in (\Phi)^{\mathfrak{T}^{I_0}_A}_{\mathcal{V}}$ (where $\mathcal{V}$ is any valuation, since Φ is closed), that is, whether Φ is true in the root of the A execution tree.

On the other hand, we know that there are several transition systems that are bisimilar to the execution tree $\mathfrak{T}^{I_0}_A$. The following theorem states that the formula evaluation in $\mu\mathcal{L}$ is indeed invariant w.r.t. bisimilarity, so we can equivalently check any such transition system.

Theorem 2.1. *Let $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be two bisimilar transition systems. For every pair of states σ_1 and σ_2 such that $\sigma_1 \sim \sigma_2$ (including the initial ones) and for all formulas Φ of $\mu\mathcal{L}$, we have that $\sigma_1 \in (\Phi)^{\mathfrak{A}_1}_{\mathcal{V}}$ iff $\sigma_2 \in (\Phi)^{\mathfrak{A}_2}_{\mathcal{V}}$.*

Proof. The proof is analogous to the standard proof of bisimulation invariance of μ-calculus [27], though taking into account our specific definition of bisimulation, which makes use of conjunctive queries and certain answers.

□

In particular, if, for some reason, we can get a *finite* transition system bisimilar to the execution tree, then the following theorem applies.

Theorem 2.2. *Checking a $\mu\mathcal{L}$ formula Φ over a finite transition system $\mathfrak{A}_A = \langle \Sigma, \sigma_0, L, \Rightarrow \rangle$ can be done in time*

$$O((|\mathfrak{A}| \cdot |\Phi|)^k)$$

where $|\mathfrak{A}| = |\Sigma| + |\Rightarrow|$, i.e., the number of states plus the number of transitions of $\mathfrak{A}$, $|\Phi|$ is the size of formula Φ (in fact, considering conjunctive queries as atomic), and k is the number of nested fixpoints, i.e., fixpoints whose variables are one within the scope of the other.

Proof. It suffices to use the standard μ-calculus model checking algorithms [62], with the proviso that for atomic formulas we use the computation of certain answers of conjunctive queries. □

Example 2.3.1. *Let us consider again our running example with initial artifact data instance I_0 of Example 2.1.1 where* $\mathsf{Customer}^{I_0} = \{\langle 337505, JohnSmith\rangle, \langle 125232, MaryStewart\rangle\}$, $\mathsf{Service}^{I_0} = \{\langle L057, 100\rangle, \langle L113, 150\rangle, \langle C002, 50\rangle\}$, *and all other relations are empty. The following liveness property checks if it is possible to obtain the provision of whatever service, i.e., whether by executing tasks we can eventually get to a state where some service has been provided:*

$$\mu Z.\ (\exists x, y.\mathsf{ServiceProvided}(x, y)\ \vee \bigvee_{T \in \mathcal{T}} \langle - \rangle - Z)$$

The formula is true, since a state where $\mathsf{ServiceProvided}(L057, 337505)$ *holds can be reached from the initial state through the following sequence of tasks:*
$\mathsf{ClaimService}(337505, L057)$,
$\mathsf{MakePayment}(337505, L057, 100)$,
$\mathsf{GrantApproval}(L057)$ *and finally*
$\mathsf{ProvideServices}()$.

Next, consider the safety property asking whether every possible reachable instance will always contain the information that the service $L113$ has been paid and provided:

$$\nu Z.\ (\exists x, y, z.\mathsf{Payment}(L113, x, y) \wedge\ \mathsf{ServiceProvided}(L113, z)\ \wedge \bigwedge_{T \in \mathcal{T}} [-](Z))$$

This is trivially false, since in the initial instance I_0 there is no payment for any service.

As a last example, we look at a fairness property, expressing that it is always true that eventually a service is provided:

$$\nu Z_1.(\mu Z_2.((\exists x_1, x_2, x_3.\mathsf{Service}(x_1, x_2) \wedge \mathsf{ServiceProvided}(x_1, x_3)) \vee \langle - \rangle - Z_2) \wedge [-]Z_1)$$

This is not the case, because there is an (infinite) path in the execution tree, e.g. the one obtained by repeating forever action $\mathsf{GrantApproval}(L113)$, *that passes through states in which* $\exists x_1, x_2, x_3.\mathsf{Service}(x_1, x_2, x_3) \wedge \mathsf{ServiceProvided}(x_1, x_3)$ *will never hold.*

More sophisticated temporal properties, such as strong forms of fairness, are also easily expressible in $\mu\mathcal{L}$. ■

2.4 Decidability of weakly-acyclic conjunctive artifacts

In this section we study decidability of verification of conjunctive artifacts. First, observe that, so far, we do not have a concrete technique for the verification problem, since the model checking results in Theorem 2.2 only apply to finite structures. In fact, as a consequence of the undecidability of the implication problem for tgds (see e.g. [2]), it is obvious that, without any restrictions on effect specifications, model checking in our setting is undecidable. Addressing sufficient conditions for decidability is the purpose of this section. We start by introducing the notion of execution transition system and showing its relationship with the execution tree.

Execution transition system. Given an artifact $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ with initial artifact data instance I_0, we define the *execution transition system* $\mathfrak{S}_A = \langle \Sigma_s, \sigma_{0,s}, L_s, \Rightarrow_s \rangle$ inductively as follows:

- $\sigma_{0,s} \in \Sigma$ and such that $L_s(\sigma_{0,s}) = I_0$;
- for all instances $\sigma \in \Sigma_s$ and for each task $T \in \mathcal{T}$ such that there exists a rule $\varrho = \pi \mapsto T$ such that $L_s(\sigma) \rhd \pi$, let $I' = do(T, L_s(\sigma))$ be data instance resulting from the execution of task T in $L_s(\sigma)$ then:
  - if there exists an instance $\sigma' \in \Sigma_s$ such that $L_s(\sigma') \overset{h}{=} I'$ then add the transition $\sigma \overset{T}{\Longrightarrow} \sigma'$ to $\Rightarrow_s$;
  - if such a state does not exists, then add the a new state $\sigma_{I'}$ to Σ with $L_s(\sigma_{I'}) = I'$ and add the transition edge $\sigma \overset{T}{\Longrightarrow} \sigma_{I'}$ to $\Rightarrow_s$.

Theorem 2.3. *Let $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ be an artifact with initial data instance I_0. The execution tree $\mathfrak{T}_{A,I_o} = \langle \Sigma_t, \sigma_{0,t}, L_t, \Rightarrow_t \rangle$ is bisimilar to the execution transition system $\mathfrak{S}_{A,I_0} = \langle \Sigma_s, \sigma_{0,s}, L_s, \Rightarrow_s \rangle$.*

Proof. Let us consider the bisimulation relation $B_{ts} = \{\langle \sigma_t, \sigma_s \rangle \mid \sigma_t \in \Sigma_t \wedge \sigma_s \in \Sigma_s \wedge L_s(\sigma_s) \overset{h}{=} L_t(\sigma_t)\}$. This is the relation formed by couples of states of the two transition systems such that their labeling data instances are homomorphically equivalent. We show that B_{ts} is a bisimulation (according to our definition). Indeed consider $\langle \sigma_s, \sigma_t \rangle \in B_{ts}$. Then:

1. For each cq, since $L_s(\sigma_s) \overset{h}{=} L_t(\sigma_t)$ we have that $cert^{L_s(\sigma_s)}(cq) = cert^{L_t(\sigma_t)}(cq)$ from the definition of certain answers and homomorphical equivalence.

2. If $\sigma_t \overset{T}{\Longrightarrow} \sigma'_t$ then there is a rule $\varrho = \pi \mapsto T$ and $L_t(\sigma_t) \rhd \pi$. Since $L_s(\sigma_s) \overset{h}{=} L_t(\sigma_t)$ then *(i)* $L_s(\sigma_s) \rhd \pi$ as well, so $\sigma_s \overset{T}{\Longrightarrow} \sigma'_s$ moreover it is easy to see that $L_t(\sigma'_t) \overset{h}{=} L_s(\sigma'_s)$ by considering definition of executing a task.

3. Symmetric to the previous case.

Finally observe that since $L_t(\sigma_{0,t}) = L_s(\sigma_{0,s}) = I_0$ we trivially get that $\langle \sigma_{0,s}, \sigma_{0,t} \rangle \in B_{ts}$.

□

This theorem basically allow us to make use of the execution transition system rather than an execution tree for our verification tasks, taking advantage of Theorem 2.1. In other words, the certain answer semantics give us the freedom of using *equivalence classes* of homomorphically equivalent instances for the purpose of verification. Notice, however, that this theorem is not sufficient to achieve a decidability result, since the number of states in the execution transition system is bounded only by the number of homomorphically non-equivalent data instances, which is infinite in general. In the following, we focus on conditions that guarantee its finiteness.

Inflationary approximate. Artifacts can both increase and decrease the size of the data stored in the data component as tasks are executed. For the development below, it is convenient to disregard the possibility of erasing data, so as to have a sort of abstraction of the original artifact in which the information monotonically increases only. To do so we introduce what we call here the *inflationary approximate* of an artifact, which is a variant of the original one in which, essentially, information only increases. Notice that we are not interested in any way to the actual behavior, i.e., transition system of the inflationary approximate. We are interested only in the fact that the inflationary approximate gives us an upper bound on the data instances constituting the state of the transition system of the original artifact. In particular, if such a bound is finite, we get that also the states of the original transition system are finite, and hence finite state model checking techniques can be used.

Given an artifact $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ let us introduce $A^+ = \langle \mathcal{D}, \mathcal{T}^+, \top \rangle$, the *inflationary approximate* of A, that differs from A in that: *(i)* every effect $T^+ \in \mathcal{T}^+$ copies all the fact and *(ii)* the rules are of the form $\varrho = \top \mapsto T^+$ for every task T^+, namely it is always possible to execute a task.

Let $\mathcal{I}$ be the set of possible interpretation over $\mathcal{S}$, in the following we will make use of two different functions: the first one, $\mathsf{f} : \mathcal{T} \times \mathcal{I} \to \mathcal{I}$, is defined as $\mathsf{f}(T, I) = do(T, I)$, so it computes the usual result of executing a task on I; while the second one, $\mathsf{g} : \mathcal{T}^+ \times \mathcal{I} \to \mathcal{I}$, is the inflationary approximate of the first one: $\mathsf{g}(T^+, I) = do(T^+, I)$, that is, it generates the result of executing the inflationary task T^+ on I. For technical reasons, we need of to compare instances that disagree on names of labeled nulls and, to do so, we make use again of the notion of *nulls renaming* introduced in Section 2.2.

Given two instances I_1 and I_2, we say that I_1 *is contained in* I_2 *modulo nulls renaming*, written as $I_1 \overset{mnr}{\subseteq} I_2$, if there exists a nulls renaming $r : I_1 \to I_2$.

Let $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ be an artifact, and $A^+ = \langle \mathcal{D}, \mathcal{T}^+, \top \rangle$ be its inflationary approximate, we introduce, for every task T^+, $I_{good} = \{I_0 \cup \mathsf{g}(T^+, I) \mid I \subseteq I_{good}\}$. Now we define the instance $I_{I_0}^{max} = \bigcap\{I_{good} \mid I_{good} = \{I_0 \cup \mathsf{g}(T^+, I) \mid I \subseteq I_{good}\}\}$.

Notice also that, as an immediate consequence of its definition, we get that $\mathsf{g}(T, I_{I_0}^{max}) = I_{I_0}^{max}$, since $I_{I_0}^{max}$ is a *fixpoint* and, indeed, the *least fixpoint* [126].

Lemma 2.1. *Let $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ be an artifact with initial data instance I_0, and let $I_{I_0}^{max}$ be as above. For every task T^+ and for every sequence of instances $I_0, \ldots, I_n$ such that $I_{i+1} = \mathsf{g}(T^+, I_i)$, we have that $I_i \overset{mnr}{\subseteq} I_{I_0}^{max}$, for $i = 0, \ldots, n$.*

Proof. By induction on the number i of application of the g function.

Base case: trivial, by definition of $I_{I_0}^{max}$ we have that $I_0 \overset{mnr}{\subseteq} I_{I_0}^{max}$.

Inductive case: we show that for every task T, if $I_{i+1} = \mathsf{g}(T, I_i)$, then $I_{i+1} \overset{mnr}{\subseteq} I_{I_0}^{max}$. Recalling that g is inflationary, we have that it is also *monotonically increasing*, namely, for every task T and instance I, we have that $I \subseteq \mathsf{g}(T, I)$. Since by inductive hypothesis $I_i \overset{mnr}{\subseteq} I_{I_0}^{max}$, we get that for every task T^+, $I_{i+1} = \mathsf{g}(T^+, I_i) \overset{mnr}{\subseteq} I_{I_0}^{max}$ holds. □

Lemma 2.2. *Let $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ be an artifact with initial data instance I_0, and let $I_{I_0}^{max}$ be as above. For every task $T \in \mathcal{T}$ and for every sequence of instances $I_0, \ldots, I_n$, such that $I_{i+1} = \mathsf{f}(T, I_i)$, we have that $I_i \overset{mnr}{\subseteq} I_{I_0}^{max}$, for $i = 0, \ldots, n$.*

Proof. By induction on the length n of application of function f.

Base case: trivial, by definition of $I_{I_0}^{max}$ we have that $I_0 \overset{mnr}{\subseteq} I_{I_0}^{max}$.

Inductive case: for the sake of readability, we split the proof in two parts: we first show that (1) for every task T and couple of instances I_f, I_g such that $I_\mathsf{f} \overset{mnr}{\subseteq} I_\mathsf{g}$, we have that $\mathsf{f}(T, I_\mathsf{f}) \overset{mnr}{\subseteq} \mathsf{g}(T, I_\mathsf{g})$ and then that (2) for every task T, if $I' = \mathsf{f}(T, I)$, then $I' \overset{mnr}{\subseteq} I_{I_0}^{max}$. Starting from (1), for every task $T \in \mathcal{T}$, let us use $I'_\mathsf{f} = \mathsf{f}(T, I_\mathsf{f})$ and $I'_\mathsf{g} = \mathsf{g}(T^+, I_\mathsf{g})$, we show how to construct a nulls renaming $r : I'_\mathsf{f} \to I'_\mathsf{g}$. Notice that $I'_\mathsf{f} = I'_{\mathsf{f},old} \cup I'_{\mathsf{f},new}$, namely, I'_f is made up by some facts that may be copied from I_f, and some new facts. On the other hand, $I'_\mathsf{g} = I_\mathsf{g} \cup I'_{\mathsf{g},new}$. Since, $I'_{\mathsf{f},old} \overset{mnr}{\subseteq} I_\mathsf{g}$ we have, by definition, that $r : I'_{\mathsf{f},old} \to I_\mathsf{g}$. We now extend r in order to cover $I'_{\mathsf{f},new}$. Any new fact $R_i^{I'_\mathsf{f}}(\vec{x}, \vec{w}, \vec{d})|_{\eta_i}^{\psi} \in I'_{\mathsf{f},new}$ comes from an effect specification $\exists \vec{y}.\phi(\vec{x}, \vec{y}, \vec{c}) \to \exists \vec{w}.\psi(\vec{x}, \vec{w}, \vec{d})$, and since the set $\vec{\eta}_\mathsf{f} = (\exists \vec{y}.\psi(\vec{x}, \vec{y}, \vec{c}))^{I_\mathsf{f}}$ are computed over the instance $I_\mathsf{f} \overset{mnr}{\subseteq} I_\mathsf{g}$, we have that (with abuse of notation) $\vec{\eta}_\mathsf{f} \overset{mnr}{\subseteq} \vec{\eta}_\mathsf{g}$ with $\vec{\eta}_\mathsf{g} = (\exists \vec{y}.\psi(\vec{x}, \vec{y}, \vec{c}))^{I_\mathsf{g}}$. Hence, for every value $\ell_{i,\mathsf{f}}$ introduced in $I'_{\mathsf{f},new}$ in a certain position of the schema, we have a correspondent value $\ell_{i,\mathsf{g}}$ in $I'_{\mathsf{g},new}$ in the same position. We then extend the identity r with $r(\ell_{i,\mathsf{f}}) = \ell_{i,\mathsf{g}}$ for each i and $r(d_j) = d_j$ for every new constant $d_j \in \vec{d}$ introduced by effects. It is easy to verify that $r : I'_\mathsf{f} \to I'_\mathsf{g}$ is indeed a nulls renaming by construction. We now prove (2): by inductive hypothesis we have $I \overset{mnr}{\subseteq} I_{I_0}^{max}$, from Lemma 2.1 we have that, for evert task $T \in \mathcal{T}$, $\mathsf{g}(T^+, I) \overset{mnr}{\subseteq} I_{I_0}^{max}$, from (1) that $\mathsf{f}(T, I) \overset{mnr}{\subseteq} \mathsf{g}(T, I)$, and therefore, by transitivity, we get the claim.

□

We thus showed that the data instances of inflationary approximate bounds the data instances of the original artifact. The next step is to find conditions that guarantee finiteness of the data instances. To do so, we resort to literature on boundedness of data exchange and the condition of weak acyclicity defined there. Before continuing, we briefly summarize such notions in the paragraph below.

Boundedness of data exchange and weak acyclicity. The *data exchange* problem addresses the issue of translating and restructuring data from one logical

schema, called *source schema* to a new one, the *target schema*. Technically, the source and the target schema are related through a set of dependencies, called source-to-target dependencies, that, intuitively, formalize how to restructure data in the target schema, while the so-called set of target-to-target dependencies is used to represent constraints on the target schema. Both these dependencies have the form of containment (or implication) between conjunctive queries. Dependencies of this form are called *tuple-generating dependencies*, or *tgds*. The problem of data exchange is then the following: given an instance of the source schema, materialize an instance over the target schema by *chasing*, i.e., recursively applying all tgds as many times as possible. However, in principle there is no guarantee that chasing will ever finish. Indeed, roughly speaking, tgds generate databases that include new "unknown" values (i.e., labelled nulls). E.g., a dependency may express the constraint on the new database that "Every employee is involved in *a* project" without telling us which project. Clearly, problems arise when such labelled null values are used for generating new ones, therefore creating a sort of loop that makes the resulting instance infinite. In order to avoid this obstacle, restriction on the form of tgds allowed have been proposed, so as to enforce the so-called *weak acyclicity*, meaning that, intuitively, dependencies should not generate values in a cyclic way, hence guaranteeing the termination of the chase and a finite resulting instance [65]. Notice that weak acyclicity is only a sufficient condition to obtain such a result and lately several generalization of the condition have been proposed [106, 58, 103, 104]. Here we stick to the original definition of weak acyclicity for simplicity, but we stress that all the results we are proposing also hold for more general conditions that guarantee the termination of the chase and the finiteness of the resulting instance.

Weakly acyclic artifacts. After this intermezzo, we are now ready to define sufficient conditions on artifact, corresponding to the above notion of weak acyclicity, that guarantee that instances generated by the inflationary approximate, and thus by the original artifact, are indeed finite.

Roughly speaking, the above lemmas guarantee that every possible instance that can be produced from I_0 by applying in every possible way $\mathtt{f}$ and $\mathtt{g}$ functions is bounded by the least fixpoint $I_{I_0}^{max}$. Notice however that $I_{I_0}^{max}$ is infinite in general, so, in order to get decidability, we need a finite bound on $I_{I_0}^{max}$. To get such condition we exploit results from [65] on *weakly-acyclic* tgds.

Weak-acyclicity is a syntactic notion that involves the so-called *dependency graph* of the set of tgds TG. Informally, a set TG of tgds is weakly-acyclic if there are no cycles in the dependency graph of TG involving "existential" relation positions. The key property of *weakly-acyclic* tgds is that chasing a data instance with them (i.e., applying them in all possible way) generates a set of facts (a database) that is finite. Formally, given an artifact $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$, the dependency graph (that is a directed graph) is constructed as follows: (1) for every relation symbol $R_i \in \mathcal{S}$ there is a node (called position) for every pair (R_i, att) where att is an attribute in R_i and (2) add edges as follows: for every action $\xi = \exists y.\phi(\vec{x}, \vec{y}, \vec{c}) \rightarrow \exists w.\psi(\vec{x}, \vec{w}, \vec{d})$ and for every $x \in \vec{x}$ that occurs in ψ: for every occurrence of x in ϕ in position p:

- for every occurrence of x in ψ in position p' add an edge $p \rightarrow p'$ (if it does not already exist);

- in addition, for existential variable $w_i \in \vec{w}$ and for every occurrence of w_i in ψ in position p'' add a *special edge* $p \xrightarrow{*} p''$ (if it does not already exists).

We say that A is weakly acyclic if the dependency graph of the effect specifications in $\mathcal{T}$ contains no cycles going trough special edges. Notice that if A is weakly acyclic, its inflationary approximate A^+ is weakly acyclic as well.

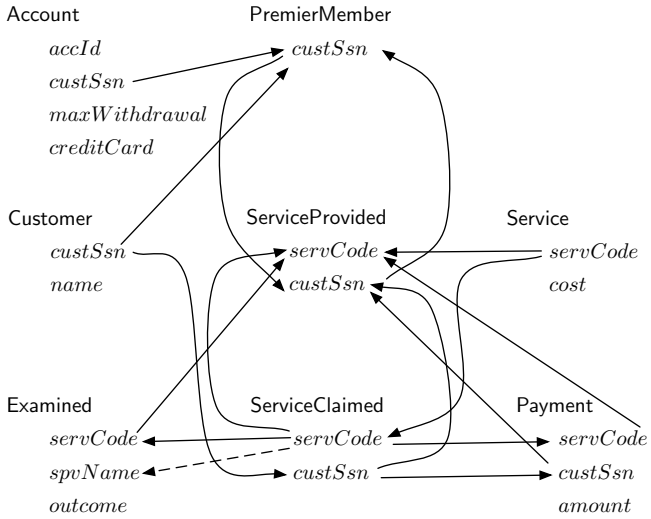


Figure 2.2. Dependency graph of the running example.

Example 2.4.1. *It is easy to see that the artifact in our running example is weakly acyclic. To verify this, we build the dependency graph associated to it, shown in Figure 2.2, and we check that there are no cycles going through special edges. In our case there is a single special edge, which is denoted by a dashed arrow, and indeed such an edge is not involved in any cycle.* ■

We would like to exploit the result of [65]. In order to do so, we show that inflationary executing a task is equivalent to a sequence of chase steps. Here we give a brief definition of chase step (more details in [65]). We first define the notion of homomorphism from a conjunctive formula $\exists y.\phi(\vec{x}, \vec{y}, \vec{c})$ to an instance I as a mapping h from the variables $\vec{x} \cup \vec{y}$ to $const(\Delta) \cup ln(\Delta)$ such that for every atom $R_i(x_1, \ldots, x_n)$ of ϕ, the fact $R_i(h(x_1), \ldots, h(x_n))$ is in I. Now we are ready to define a *chase step*: let I be an instance, $\xi = \exists y.\phi(\vec{x}, \vec{y}, \vec{c}) \rightarrow \exists w.\psi(\vec{x}, \vec{w}, \vec{d})$ an effect specification, i.e., a tgd, and h an homomorphism from $\exists y.\phi(\vec{x}, \vec{y}, \vec{c})$ to I such that there is no extension of h to an homomorphism h' from $\exists y.\phi(\vec{x}, \vec{y}, \vec{c}) \wedge \exists w.\psi(\vec{x}, \vec{w}, \vec{d})$ to I. We say that ξ *can be applied to* I *with homomorphism* h. Let I' be the union of I with the set of facts obtained by: (a) extending h to h' such that each variable in y is assigned a fresh labeled null, followed by (b) taking the image of the atoms of ψ under h'. We say that the result of applying ξ to I with h is I', and we write $I \xrightarrow{\xi,h} I'$. To conclude, a *chase sequence* of I_0 with a set $\vec{\xi}$ of effect specifications, namely, tgds, is a sequence (finite or infinite) of chase step $I_i \xrightarrow{\xi_i,h_i} Ii+1$ with $i = 0, 1, \ldots$ and $\xi_i \in \vec{\xi}$.

We are now ready to define a correspondence between the enactment of a task and a chase sequence.

Lemma 2.3. *Let A be an artifact, $A^+ = \langle \mathcal{D}, \mathcal{T}^+, \top \rangle$ its inflationary approximate, and I, J two instances such that $I \overset{mnr}{=} J$. For every task $T^+ \in \mathcal{T}^+$, if $I \overset{T^+}{\Longrightarrow} I'$ then there exists a chase sequence (possibly empty) $J \xrightarrow{\xi_i,h_i}, \ldots, \xrightarrow{\xi_j,h_j} J'$ with $\xi_i, \ldots, \xi_j \in \mathcal{T}^+$ such that $I' \overset{mnr}{=} J'$.*

Proof. Recall that enacting a task in $T^+ \in \mathcal{T}^+$ at state σ such that $L_s(\sigma) = I$, that is, execute the $\mathsf{g}(T^+, I)$ function, is inflationary, and so is the chase step: therefore, the resulting instance, say I', is $I' = I \cup I'_{new}$ where I'_{new} is the set of "new" facts just added by T^+. Nevertheless, I'_{new} may be the empty set. In this case, the enactment of a task can still be performed (resulting in $\mathsf{g}(T^+, I) = I$) while the chase step cannot be performed (no chase step involving tgds in T^+ can be performed, since every homomorphism from ϕ to J can be extended from $\phi \wedge \psi$). This is a simple consequence of the definition of chase step, g function and execution transition system. But if $I'_{new} = \emptyset$, no new facts are added, and so no chase steps are needed. Stepping back to the general and more interesting case, by definition of execution of a task, I'_{new} is made up by facts of the form $\psi_i^{I'}(\vec{x}, \vec{w}, \vec{d})|_{\eta_{i,j}}$ for each effect specification $\xi_i = \exists \vec{y}.\phi_i(\vec{x}, \vec{y}, \vec{c}) \rightarrow \exists \vec{w}.\psi_i(\vec{x}, \vec{w}, \vec{d})$ in T^+ and each assignment $\eta_{i,j} \in \vec{\eta_i}$ where $\vec{\eta_i} = \exists \vec{y}.\phi_i(\vec{x}, \vec{y}, \vec{c})^I$. Since $I \overset{mnr}{=} J$, there exists $r : I \rightarrow J$. For each $\eta_{i,j}$ let us consider the function $h_{i,j}$ built in this way: *(i)* for each $x \in \vec{x}$ (resp. $y \in \vec{y}$) such that $\eta_{i,j}(x) = c$ (resp. $\eta_{i,j}(y) = c$) with $c \in const(\Delta)$, $h_{i,j}(x) = \eta_{i,j}(x)$ (resp. $h_{i,j}(y) = \eta_{i,j}(y)$); *(ii)* for each $x \in \vec{x}$ (resp. $y \in \vec{y}$) such that $\eta_{i,j}(x) = \ell$ (resp. $\eta_{i,j}(y) = \ell$) with $\ell \in ln(\Delta)$, $h_{i,j}(x) = r(\eta_{i,j}(x))$ (resp. $h_{i,j}(y) = r(\eta_{i,j}(y))$). We have that $h_{i,j}$ is actually an homomorphism from the set of variables $\vec{x}$, $\vec{y}$ in ϕ_i to the instance J. Since the set of new facts coming from effect ξ_i can be identified by the couple $\langle \xi_i, \eta_{i,j} \rangle$, let us informally write $I'_{new} = \{\langle \xi_1, \eta_{1,1} \rangle, \ldots, \langle \xi_n, \eta_{n,m} \rangle\}$.

Now we show that $J \xrightarrow{\xi_1,h_{1,1}}, \ldots, \xrightarrow{\xi_n,h_{n,m}} J'$ is the chase step sequence we need in order to get the claim (with each $h_{i,j}$ obtained from $\eta_{i,j}$ and r as before). Let us label such tuples (and resulting instances) with consecutive numbers. We get $J \xrightarrow{0} J_1 \xrightarrow{1}, \ldots, \xrightarrow{p} J'$. We prove by induction that, in any instance J_i with $0 \leq i < p$ it is possible to perform the $i+1$-th chase step, and then that $I' \overset{mnr}{=} J'$.

Base case: Since $\langle \xi_1, \eta_{1,1} \rangle$ is in I_{new} this means that $h_{1,1}$ is an homomorphism from variables in ψ_1 to J and that it cannot be extended to an homomorphism $\phi_1 \wedge \psi_1$ to J (otherwise $\langle \xi_1, \eta_{1,1} \rangle$ would not result in a new fact in I'). So the first chase step can be executed.

Inductive case: By inductive hypothesis every (and only) chase steps labeled with numbers less than i have been executed ($I_0 \xrightarrow{0} J_1 \xrightarrow{1} J_2 \xrightarrow{2} \ldots \xrightarrow{i} J_{i+1}$). Now we prove that it is possible to perform $J_{i+1} \xrightarrow{i+1} J_{i+2}$. If the couple labeled with $i+1$, say, $\langle \xi_{i+1}, \eta_{i+1,j} \rangle$, is in I'_{new} this means that $h_{i+1,j}$ is an homomorphism from variables in ψ_{i+1} to J and that it cannot be extended to an homomorphism $\phi_{i+1} \wedge \psi_{i+1}$ to J. Since the chase is inflationary, we have that $h_{i+1,j}$ is also an homomorphism from ψ_{i+1} to J_{i+1}, and moreover, by inductive hypothesis, the $i+1$-th couple has not been used in a previous chase step, so $h_{i+1,j}$ cannot be extended from $\phi_{i+1} \wedge \psi_{i+1}$ to J_{i+1}, so the chase step $J_{i+1} \xrightarrow{\xi_{i+1},\eta_{i+1,j}} J_{i+2}$ can be performed. Since $J' = J \cup J'_{new}$ and, by construction, $I'_{new} \overset{mnr}{=} J'_{new}$, we get that $I' \overset{mnr}{=} J'$.

□

Lemma 2.4. *Let $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ be a weakly acyclic artifact with initial data instance I_0, and $A^+ = \{\mathcal{D}, \mathcal{T}, \mathcal{C}\}$ its inflationary approximate. The fixpoint $I_{I_0}^{max}$ has finite cardinality.*

Proof. Let $\mathfrak{S}_{A^+} = \langle \Sigma_s, \sigma_{0,s}, L_s, Tr_s \rangle$ the execution transition system of A^+. Roughly speaking, $\mathfrak{S}_{A^+}$ is the transition system obtained by applying the $\mathtt{g}$ function in every possible way, and generating instances that are homomorphically non-equivalent only. We show that for every sequence $\sigma_{0,s} \overset{T_i}{\Longrightarrow} \ldots \overset{T_j}{\Longrightarrow} \sigma_{n,s}$, there exists a sequence of chase steps $I_0 \xrightarrow{\xi_i, h_i} \ldots, \xrightarrow{\xi_j, h_j} I_m$, where $I_0 = L_s(\sigma_{0,s})$ and $\xi_i \ldots \xi_j$ are tgds in $\mathcal{T}^+$, such that $L_s(\sigma_{i,s}) \overset{mnr}{=} I_j$. By induction on the number of task enactments.

Base case: trivial, since the equality modulo nulls renaming is reflexive.

Inductive case: by inductive hypothesis, there exists a $k \leq n$ and a $p \leq m$ such that $L_s(\sigma_{k,s}) \overset{mnr}{=} I_p$. We show that if $\sigma_{k,s} \overset{T}{\Longrightarrow} \sigma_{k+1,s}$ then there exists a sequence of chase step $I_p, \xrightarrow{\xi_i, h_i} \ldots \xrightarrow{\xi_j, h_j} I_{p+\ell}$ such that $L_s(\sigma_{k+1,s}) \overset{mnr}{=} I_{p+\ell}$. This is guaranteed from Lemma 2.3.

We proved that every instance generated by the execution transition system is equal modulo nulls renaming to an instance generated by a sequence of chase step. By results in [65] we have that if effect specifications in $\mathcal{T}^+$ are weakly acyclic, then there exists a polynomial in the size of the initial instance I_0 that bounds the length, and so the size, of every chase sequence of I_0 with $\mathcal{T}^+$. Since two instances that are equals modulo nulls renaming have also the same size, results in [65] also apply to our (inflationary) execution transition system, so $I_{I_0}^{max}$ has finite cardinality. □

Theorem 2.4. *Let $A = \langle \mathcal{D}, \mathcal{T}, \mathcal{C} \rangle$ be a weakly acyclic artifact with initial data instance I_0. For every formula Φ of $\mu\mathcal{L}$, verifying that Φ holds in A with initial data instance I_0 is decidable.*

Proof. By Theorem 2.3 and Theorem 2.1, we can perform model checking of Φ on the execution transition system for A. Now, by Lemma 2.2, we have that all data instances that can be assigned to the states of the execution transition system for A must be subsets of $I_{I_0}^{max}$. By Lemma 2.4, we get that $I_{I_0}^{max}$ has a finite cardinality. This implies that execution transition system is finite and Theorem 2.2 can be applied. □

As mentioned above, all these results can be readily extended to generalization of the weak acyclicity condition as those proposed in [58, 106, 103, 104].

2.5 Discussion

We presented a framework for modeling and verifying artifact-centric systems that is the first to explore the connection between dynamic systems and the area of data exchange and integration. This works put the basis for a promising line of research that will be investigated throughout this thesis, and that inspired other works in the community. Related works will be discussed in Section 3.6, at the end of next Chapter.

We briefly comment on the significance of weak acyclic conditions (the original one as well as its extensions mentioned above). We argue that the restriction is not too severe, and that in most real cases artifacts are indeed weakly acyclic or can be transformed into weakly acyclic ones at cost of redesign. Our argumentation is grounded on the following observation: if an artifact is not weakly acyclic, then it will repeatedly generate new values from the old ones. Such values will depend on a chain of previous values of *unbounded* length meaning that they depend on old values that are arbitrarily far in the past, and moreover on an unbounded number of such old values. Notice that, if such a number can be bounded, then, in principle, the artifact can be rewritten into a weakly acyclic one. While such unbounded systems exist in theory, e.g., Turing machines where the artifact data component is the tape, most services, that are naturally more abstract than Turing machines, does not require such an unboundedness in practice. On the other hand, while we believe that most services can be rewritten into weakly acyclic ones, how to systematically to this transformation is an issue that requires further studies.

Chapter 3

Relational artifact-centric processes

One of the limitations of the approach in the previous chapter is the limited expressivity of tasks. Indeed, recalling that tasks are defined using conjunctive queries, the strongest contraints is that negation is not allowed. In this chapter we overcome this limitation by introducing negation in specifications of tasks. This gain in expressivity comes with the cost of giving up all the theory of the incomplete information.

We consider several artifacts (fixed in advance) forming a so called *relational artifact system*, each constituted by a *relational database* evolving over time. Evolution of artifact is now refined, and not only defined by the process but also by stating dynamic properties in terms of *intra-artifact* and *inter-artifact dynamic constraints*. We express such constraints, and other dynamic properties of interest, in a suitable variant of μ-calculus.

We are interested in two main reasoning tasks. The first one is *conformance of a process to an artifact system*, which consists in checking whether a given process generates the correct lifecycle for the various artifacts and, more generally, whether it satisfies all intra-artifact and inter-artifact constraints. The second reasoning task is *process verification*, that is, checking whether a process (over an artifact system) verifies general dynamic properties of interest. Both these reasoning tasks in principle can be based on model checking, though, we have to deal with potentially infinite states. We ground on the result of the previous chapter and we identify a class of *weakly acyclic* processes for which model checking is decidable. Under such a restriction, we are guaranteed that the number of new objects introduced by the execution of actions is finite, and hence, the whole process is finite-state.

3.1 The framework

Relational artifact. A relational artifact is a relational database evolving over time. Hence, it is characterized by the usual notions of *database schema*, giving the structure of data, and *database instance*, detailing the actual data contained in it. Furthermore, it is augmented by a set of *intra-artifact dynamic constrains*, that are temporal constraints expressed in the temporal logic $\mu\mathcal{L}$ introduced later, which

allows us to express various constraints over the database: we can assert the usual ones, such as inclusion dependencies, which now become safety temporal constrains, and also what is typically called the *artifact lifecycle*, namely, dynamic constrains on the sequencing of configurations the database may pass through.

More formally, a *relational artifact* is a tuple $A = \langle \boldsymbol{R}, I_0, \Phi \rangle$ where:

- $\boldsymbol{R} = \{R_1, \ldots, R_n\}$ is a database schema, constituted by a set of relation schemas;
- I_0 is a database instance, compliant with the schema $\boldsymbol{R}$, which represents the initial state of the artifact;
- Φ is a $\mu\mathcal{L}$ formula over $\boldsymbol{R}$ constituted by the conjunction of all intra-artifact dynamic constraints of A.

Notice that if we project the dynamic formula Φ over the initial artifact instance I_0, we may get (depending on the structure of Φ) static, i.e., local, constraints on I_0. From now on, we assume to deal with *well-formed artifacts*, namely, artifacts whose initial instance satisfies such local constraints.

Relational artifact system. A relational artifact system is composed by several relational artifacts in execution at the same time, each consisting of a database and a set of intra-artifact dynamic constraints. The dynamic interaction between them is regulated through additional constraints, also expressed in $\mu\mathcal{L}$, which we call *inter-artifact-dynamic constraints*.

We make the assumption that artifacts cannot be created or destroyed during the evolution of the system. Under such an assumption we get quite interesting undecidability and decidability results. For this reason we start with a finite set of artifacts, and over the whole evolution of the system these will remain the only ones of interest. If an artifact has a terminating lifecycle it becomes dormant, but it will persist in the system.

Formally, an *artifact system* is a pair $\mathcal{A} = \langle \{A_1, \ldots, A_n\}\ \Phi_{inter} \rangle$, where $\{A_1, \ldots, A_n\}$ is the finite set of artifacts of the system (each with its own database and intra-artifact dynamic constraints expressed in $\mu\mathcal{L}$) and Φ_{inter} is a $\mu\mathcal{L}$ formula expressing the conjunction of inter-artifact dynamic constraints.

To distinguish relations of various artifacts in $\mathcal{A}$ we use the usual *dot notation* of object-orientation, hence, a relation R_j of artifact A_i of $\mathcal{A}$ is denoted by $A_i.R_j$. When clear from the context, we drop the artifact A_i and we use R_j for the relation. We denote by $\mathcal{I}_0$ the disjoint union of all initial instances of the artifacts in $\mathcal{A}$, i.e., $\mathcal{I}_0 = \bigcup_{i=1,\ldots,n} I_{0,i}$. More generally, $\mathcal{I}$ represents the instance obtained by the (disjoint) union of the current instances of each artifact in $\mathcal{A}$.

Given a database instance I, we denote by $\mathcal{C}_I$ the active domain of I, i.e., the set of individuals (typically constants) appearing in I. Hence, the active domain of $\mathcal{I}_0$ is $\mathcal{C}_{\mathcal{I}_0}$, which is made up by all constants appearing in the initial instances of the various artifacts in $\mathcal{A}$.

Notice that, as in the previous chapter, artifacts do not include a predefined mechanism for progression. Progression is due to the execution of actions, tasks, or

services over the system, according to a given process that we will introduce later on. Here it is sufficient to assume that a progression mechanism exists, and its execution results in moving from the initial state, given by the instance $\mathcal{I}_0$, to the next one, and so on. In this way we build a *transition system* $\mathfrak{A}$, whose states represent possible system instances, and each transition an atomic step in the progression mechanism (whatever it is). In principle, we can model-check such a transition system to verify dynamic properties [43], that is exactly what we are going to do next. However, one has to consider that, in general, $\mathfrak{A}$ is infinite, hence the classical results on model checking [43, 61], that are developed for finite transition systems, do not apply. We find interesting conditions under which such a transition system is finite.

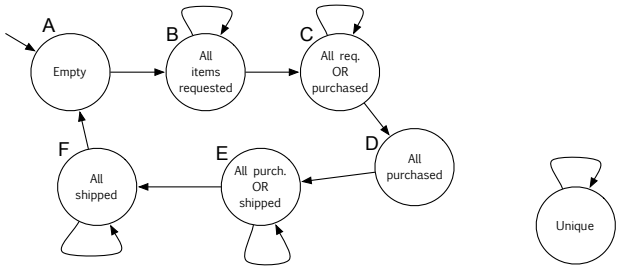
Example 3.1.1. *We model the process of purchasing items within a company. In particular, when a company's employee, that assumes the role of a* requester*, wants to purchase some items, he has to turn to a* buyer*, also internal to the company, who is responsible for purchasing such items from external* suppliers*. In our scenario, we have five actors: two requesters (*Bob *and* Alice*), a buyer (*Trudy*) and two suppliers (*SupplierA *and* SupplierB*). The whole purchasing process works as follows: in a first phase, the requester has to fill a so-called* requisition order *with some* line items *chosen from a catalogue. In our simple example the catalogue contains only a* monitor*, a* mouse *and a* keyboard*. Once the requester has completed this process, he sends the order to the buyer, which extracts the line items from it, and purchases each of them separately. In particular, the buyer groups together into a* procurement order *line items (belonging to a requisition order) that will be purchased from a particular supplier. As a result of this phase, we get different procurement orders, each containing line items that the buyer requests from a single supplier. Then the supplier ships back to the buyer the items included in the procurement order he received, and finally, the items are delivered to the original requester. Of course, we can have many orders processed simultaneously in the system, although we will impose some restrictions.*

In this example, we consider the relational artifact system $\mathcal{A} = \langle\{\mathsf{ReqOrders}, \mathsf{ProcOrders}\}, \Phi_{inter}\rangle$ *containing two relational artifacts, holding all relevant data about requisition orders and procurement orders in the system.*

$\mathsf{ReqOrders} = \langle \boldsymbol{R}_{RO}, I_{0,RO}, \Phi_{RO}\rangle$

- $\boldsymbol{R}_{RO} = \{\ \mathsf{RO}(RoCode, ReqName, BuName),\ \mathsf{ROItem}(RoCode, ProdName, Status),$
  $\mathsf{Requester}(ReqName),\ \mathsf{LineItem}(ProdName, Price),$
  $\mathsf{Buyer}(BuName),\ \mathsf{Status}(StatusName)\ \}$

  A requisition order is meant to hold the data associated to every pending *requisition order: indeed, as soon as the items are delivered to the corresponding requester, each information associated to them is removed from the system. Relation* $\mathsf{RO}(RoCode, ReqName, BuName)$ *holds basic information associated to a single order i.e., order's code and both requester's and buyer's names. The requested items are kept in the relation* $\mathsf{ROItem}(RoCode, ProdName, Status)$*, whose attribute Status keeps track of the status of each line item included in the order (it can be either* requested*,* purchased *or* shipped*). Relations* $\mathsf{Requester}(ReqName)$*,* $\mathsf{LineItem}(ProdName, Price)$*,* $\mathsf{Buyer}(BuName)$ *and* $\mathsf{Status}(StatusName)$ *are included in the schema for technical convenience; in particular, the relation* Status *is needed in order to easily bind values of the attribute Status of each line item in an order.*



(a) (b)

Figure 3.1. Informal representation of *dynamic* intra-artifact constraints

- $I_{0,RO} = \{$ Requester(Bob), Requester(Alice), Buyer(Trudy), Status(requested), Status(purchased), Status(shipped), LineItem(keyboard, 20), LineItem(mouse, 10), LineItem(monitor, 200) $\}$

  According to the previous description of this example scenario, in the initial instance we only have data referred to existing requesters, buyers, suppliers and the catalogue, featuring three line items. There are no pending orders.

- *As for intra-artifact constraints, here we only give an intuition of what will be presented formally later. We want to trace the status of an ordered line item through the attribute* ROItem.*Status, so we express constraints on the evolutions of all orders in the system by relying on this attribute, as informally depicted in Figure 3.1(a). Intuitively, at the beginning, we do not have any order placed by requesters: in the current situation (henceforth called* phase*) the relations* RO *and* ROItem *are empty* $[A]$*. As orders are placed, and new requisition orders are created, we will have a phase in which all currently pending orders have status* requested $[B]$*, and such condition will hold until,* eventually*, some item in such status will be purchased by the buyer by creating procurement orders to send to suppliers, hence changing status to* purchased*. At this point, we will be in a phase such that all items belonging to existing orders are either requested or purchased* $[C]$ *and finally, at some point, all orders will be purchased* $[D]$*. Having all procurement orders sent to suppliers, some of them will be shipped back, i.e., setting the status of corresponding items to* shipped $[E]$*, and at the end, all of them will be shipped back to the buyer* $[F]$*. Finally, as the items are delivered to the requester, they will be removed from the system and the initial condition will be eventually met again. Notice that we are imposing some restrictions on the evolution of the artifact: for instance, we will not allow for creating new requisition orders (for ordering new line items) as soon as all the existing ones have been purchased* $[D]$*. Notice also that we will need a way to force the system to eventually exit self-loops. Moreover, in addition to such* dynamic *constraints, we also have some* static *ones, such as inclusion dependencies.*

ProcOrders $= \langle \boldsymbol{R}_{PO}, I_{0,PO}, \Phi_{PO} \rangle$

- $\boldsymbol{R}_{PO} = \{$ PO($PoCode, RoCode, SupName$), POItem($PoCode, RoCode, ProdName$), Supplier($SupName$), LineItem($ProdName, Price$)$\}$.

Recall that all line items assigned to the same procurement order must belong to the same requisition order. Hence, similarly to requisition orders, a procurement order's schema includes a relation PO(*PoCode*, *RoCode*, *SupName*) *holding its code, the code of the corresponding requisition order and the name of the chosen supplier. Relation* POItem(*PoCode*, *RoCode*, *ProdName*) *holds instead the set of line items in each procurement order. Attribute RoCode is replicated in this relation for convenience.* Supplier(*SupName*) *keeps the set of existing suppliers whereas* LineItem(*ProdName*, *Price*) *is the same as the one in requisition order artifact.*

- $I_{0,PO} = \{$ LineItem(keyboard, 20), LineItem(mouse, 10), LineItem(monitor, 200), Supplier(SupplierA), Supplier(SupplierB)$\}$.
- *In this example we don't want to constrain the dynamic evolution of the artifact so, informally, the only intra-artifact constraints we will consider are those needed for consistency.* ■

We now concentrate on progression mechanisms for relational artifact systems. In particular, we specify such a mechanism in terms of one or more *processes* that use actions as atomic steps. *Actions* represent atomic tasks or services that act over the artifacts and make them evolve.

Actions. We generalize the notion of effects in Chapter 2 by including negation, arbitrary quantification in preconditions, and the generation of new terms, through the use of *Skolem functions* in postconditions. Notice that, while it is conceivable that most of the actions will act on one artifact only, we do not make such a restriction. Indeed our actions are generally inter-artifact, which let us easily account for synchronisation between artifacts.

An *action* ρ for $\mathcal{A}$ has the form

$$\rho(p_1, \ldots, p_m) : \{e_1, \ldots, e_m\} \qquad \text{where:}$$

- $\rho(p_1, \ldots, p_m)$ is the *signature* of the action, constituted by a name ρ and a sequence $p_1, \ldots, p_m$ of *input parameters* that need to be substituted by constants for the execution of the action, and
- $\{e_1, \ldots, e_m\}$ is a set of effects, called the *effects' specification*.

We denote by σ a (ground) substitution for the input parameters with terms not involving variables. Given such a substitution σ, we denote by $\rho\sigma$ the action with actual parameters. All effects in the effects' specification are assumed to take place simultaneously. Specifically, an *effect* e_i has the form

$$q_i^+ \wedge Q_i^- \rightsquigarrow I_i' \qquad \text{where:}$$

- $q_i^+ \wedge Q_i^-$ is a query whose terms are variables $\vec{x}$, action parameters, and constants from $\mathcal{C}_{\mathcal{I}_0}$. The query q_i^+ is a UCQ, and the query Q_i^-, is an arbitrary FOL formula whose free variables are included in those of q_i^+. Intuitively, q_i^+ selects the tuples to instantiate the effect, and Q_i^- filters away some of them.

- I'_i is a set of facts for the artifacts in $\mathcal{A}$, which includes as terms: terms in $\mathcal{C}_{\mathcal{I}_0}$, input parameters, free variables of q_i^+, and, in addition, terms formed by applying an arbitrary Skolem function to one of the previous kinds of terms. Such Skolem terms are used as witnesses of values chosen by the external user/environment when executing the action. Notice that different effects can share a same Skolem function.

Given an instance $\mathcal{I}$ of $\mathcal{A}$, an effect e_i as above, and a substitution σ for the parameters of e_i, the effect e_i extracts from $\mathcal{I}$ the set $ans((q_i^+ \wedge Q_i^-)\sigma, \mathcal{I})$ of tuples of terms, and for each such tuple θ asserts the set $I'_i\sigma\theta$ of facts obtained from $I'_i\sigma$ by applying the substitution θ for the free variables of q_i^+. In particular, in the resulting set of facts we may have terms of the form $f(\vec{t})\sigma\theta$ where $\vec{t}$ is a set of terms that may be either free variables in $\vec{x}$, parameters, or terms in $\mathcal{C}_{\mathcal{I}_0}$. We denote by $e_i\sigma(\mathcal{I})$ the overall set of facts, i.e., $e_i\sigma(\mathcal{I}) = \bigcup_{\theta \in ans((q_i^+ \wedge Q_i^-)\sigma, \mathcal{I})} I'_i\sigma\theta$. The overall *effect* of the action ρ with parameter substitution σ over $\mathcal{I}$ is a new instance $\mathcal{I}' = do(\rho\sigma, \mathcal{I}) = \bigcup_{1 \leq i \leq m} e_i\sigma(\mathcal{I})$ for A.

Some observations are in order: *(i)* in the formalization above actions are *deterministic*, in the sense that, given an instance $\mathcal{I}$ of $\mathcal{A}$ and a substitution σ for the parameters of an action ρ, there is a *single* instance $\mathcal{I}'$ that is obtained as the result of executing ρ in $\mathcal{I}$. This is different from the framework in Chapter 2 where determinism is not enforced, as, in principle, nulls can be different from each others. However, we do not have equality in the conjunctive setting and this entails nulls are indistinguishable. *(ii)* The effects of an action are naturally a form of update of the previous state, and not of belief revision [86]. That is, we never learn new facts on the state in which an action is executed, but only on the state resulting from the action execution. *(iii)* We do not make any persistence (or frame) assumption in our formalization [118]. In principle at every move we substitute the whole old state, i.e., instance, $\mathcal{I}$, with a new one, $\mathcal{I}'$. On the other hand, it should be clear that we can easily write effect specifications that *copy* big chunks of the old state into the new one. For example, $R_i(\vec{x}) \rightsquigarrow R_i(\vec{x})$ copies the entire set of assertions involving the relation R_i. Notice that *(ii)* and *(iii)* holds in the conjunctive setting presented in the previous chapter as well.

Processes. Essentially processes are (possibly nondeterministic) programs that use artifacts in $\mathcal{A}$ to store their (intermediate and final) computation results, and use actions in $\boldsymbol{\rho}$ as atomic instructions. We assume that at every time the current instance $\mathcal{I}$ can be arbitrarily queried through the query answering services, while it can be updated only through the actions in $\boldsymbol{\rho}$. Notice that, while we require the execution of actions to be sequential, we do not impose any such constraints on processes, which in principle can be formed by several concurrent branches, including fork, join, and so on. Concurrency is to be interpreted by interleaving, as often done in formal verification [43, 61]. There can be many ways to provide the control flow specification for processes for $\mathcal{A}$. Following Chapter 2, we adopt a rule-based mechanism, but, of course, our results can be immediately generalized to any process formalism whose processes control flow is finite-state. Notice also that again, the transition system associated to a process over an artifact might not be

finite-state, since its state is formed by both the *control flow* state of the process and the *data* in the artifact system, which are in general unbounded.

Formally, a process Π over a relational artifact system $\mathcal{A}$ is a pair $\langle \boldsymbol{\rho}, \boldsymbol{\pi} \rangle$, where $\boldsymbol{\rho}$ is a finite set of actions and $\boldsymbol{\pi}$ is a finite set of condition-action rules.

A *condition-action rule* π in $\boldsymbol{\pi}$ is an expression of the form

$$Q \mapsto \rho,$$

where ρ is an action in $\boldsymbol{\rho}$ and Q is a FOL formula over artifacts' relations whose free variables are exactly the parameters of ρ, and whose other terms can be either quantified variables or terms in $\mathcal{C}_{\mathcal{I}_0}$. Such a rule has the following semantics: for each tuple σ for which condition Q holds, the action ρ with actual parameters σ *can* be executed. If ρ has no parameters then Q will be a boolean formula. Observe that processes don't force the execution of actions but constrain them: the user of the process will be able to choose any of the actions that the rules forming the process allow.

3.2 Relational artifact execution

The *execution* of a process Π over a relational artifact system $\mathcal{A}$ is defined as follows: we start from $\mathcal{I}_0$, and for each rule $Q \mapsto \rho$ in Π, we evaluate Q, and for each tuple σ returned, we execute $\rho\sigma$, obtaining a new instance $\mathcal{I}' = do(\rho\sigma, \mathcal{I}_0)$, and so on. In this way we build a *transition system* $\Upsilon(\Pi, \mathcal{A})$ whose states represent possible system instances, and where each transition represents the execution of an instantiated action that is allowed according to the process. A transition $\mathcal{I} \Rightarrow_{\Upsilon(\Pi,\mathcal{A})} \mathcal{I}'$ holds iff there exists a rule $Q \mapsto \rho$ in Π such that there exists a $\sigma \in ans(Q, \mathcal{I})$ and $\mathcal{I}' = do(\rho\sigma, \mathcal{I})$. That is, there exist a rule in Π that can fire on $\mathcal{I}$ and produce an instantiated action $\rho\sigma$, which applied on $\mathcal{I}$, results in $\mathcal{I}'$.

The transition system $\Upsilon(\Pi, \mathcal{A})$ captures the behavior of the process Π over the whole system $\mathcal{A}$.

Notice that the execution semantics of relational artifact systems is essentially the same execution semantics of conjunctive artifact systems. Indeed parameters in actions are just a more elegant and concise way to describe possible actions' (or tasks) executions, and the transition system $\Upsilon(\Pi, \mathcal{A})$ is built in a very similar way to the execution transition system $\mathfrak{S}_A$ of a conjunctive artifact A.

3.3 Dynamic constraints formalism

We turn to the dynamic constraints formalism used both to specify intra and inter dynamic constraints of artifact systems (including artifact lifecycles) and to specify dynamic properties of processes running over relational artifact systems.

As in Chapter 2, we focus on a variant of μ-calculus called $\mu\mathcal{L}$ that conforms with the basic assumption of our formalism, namely the use of range-restricted FOL queries, i.e., open formulas over a fixed set of constants, to talk about the information contained in the instances.

Formally, $\mu\mathcal{L}$ formulas over $\mathcal{A}$ have the form

$$
\begin{array}{ll}
(\neg\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \Sigma_{\mathfrak{A}} - (\Phi)^{\mathfrak{A}}_{\mathcal{V}} \\
(\Phi_1 \wedge \Phi_2)^{\mathfrak{A}}_{\mathcal{V}} & = (\Phi_1)^{\mathfrak{A}}_{\mathcal{V}} \cap (\Phi_2)^{\mathfrak{A}}_{\mathcal{V}} \\
(\Phi_1 \vee \Phi_2)^{\mathfrak{A}}_{\mathcal{V}} & = (\Phi_1)^{\mathfrak{A}}_{\mathcal{V}} \cup (\Phi_2)^{\mathfrak{A}}_{\mathcal{V}} \\
(\exists x \in \mathcal{C}_{\mathcal{I}_0}.\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \bigcup\{(\Phi)^{\mathfrak{A}}_{\mathcal{V}[x/c]} \mid c \in \mathcal{C}_{\mathcal{I}_0}\} \\
(\forall x \in \mathcal{C}_{\mathcal{I}_0}.\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \bigcap\{(\Phi)^{\mathfrak{A}}_{\mathcal{V}[x/c]} \mid c \in \mathcal{C}_{\mathcal{I}_0}\}
\end{array}
\qquad
\begin{array}{ll}
(Z)^{\mathfrak{A}}_{\mathcal{V}} & = Z\nu \subseteq \Sigma_{\mathfrak{A}} \\
(Q)^{\mathfrak{A}}_{\mathcal{V}} & = \{\mathcal{I} \in \Sigma_{\mathfrak{A}} \mid ans(Q\nu, \mathcal{I})\} \\
(\Diamond\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \{\mathcal{I} \in \Sigma_{\mathfrak{A}} \mid \exists \mathcal{I}'.\ \mathcal{I} \Rightarrow_{\mathfrak{A}} \mathcal{I}' \text{ and } \mathcal{I}' \in (\Phi)^{\mathfrak{A}}_{\mathcal{V}}\} \\
(\Box\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \{\mathcal{I} \in \Sigma_{\mathfrak{A}} \mid \forall \mathcal{I}'.\ \mathcal{I} \Rightarrow_{\mathfrak{A}} \mathcal{I}' \text{ implies } I' \in (\Phi)^{\mathfrak{A}}_{\mathcal{V}}\} \\
(\mu Z.\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \bigcap\{\mathcal{E} \subseteq \Sigma_{\mathfrak{A}} \mid (\Phi)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]} \subseteq \mathcal{E}\} \\
(\nu Z.\Phi)^{\mathfrak{A}}_{\mathcal{V}} & = \bigcup\{\mathcal{E} \subseteq \Sigma_{\mathfrak{A}} \mid \mathcal{E} \subseteq (\Phi)^{\mathfrak{A}}_{\mathcal{V}[Z/\mathcal{E}]}\}
\end{array}
$$

Figure 3.2. Semantics of $\mu\mathcal{L}$ formulas

$$
\begin{array}{rl}
\Phi ::= & Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \exists x \in \mathcal{C}_{\mathcal{I}_0}.\Phi \mid \forall x \in \mathcal{C}_{\mathcal{I}_0}.\Phi \mid \\
& \Box\Phi \mid \Diamond\Phi \mid \mu Z.\Phi \mid \nu Z.\Phi \mid Z,
\end{array}
$$

where Q is a possibly open FOL formula over the relations in the artifacts of $\mathcal{A}$, and Z is a second order predicate variable.

To define the meaning of a $\mu\mathcal{L}$ formula over an artifact system, we resort to transition systems. Let $\mathfrak{A}$ be a transition system generated by a given progression mechanism over the artifact system $\mathcal{A}$. We denote by $\Sigma_{\mathfrak{A}}$ the states of $\mathfrak{A}$, and by $\mathcal{C}_{\mathfrak{A}}$ all terms (which are in general infinite) occurring in any state of $\mathfrak{A}$. Notice that trivially $\mathcal{C}_{\mathcal{I}_0} \subseteq \mathcal{C}_{\mathfrak{A}}$. Let $\mathcal{V}$ be a predicate and individual variable valuation on $\mathfrak{A}$, i.e., a mapping from the predicate variables Z to subsets of the states $\Sigma_{\mathfrak{A}}$, and from individual variables to constants in $\mathcal{C}_A$. Then, we assign meaning to $\mu\mathcal{L}$ formulas by associating to $\mathfrak{A}$ and $\mathcal{V}$ an *extension function* $(\cdot)^{\mathfrak{A}}_{\mathcal{V}}$, which maps $\mu\mathcal{L}$ formulas to subsets of $\Sigma_{\mathfrak{A}}$. The extension function $(\cdot)^{\mathfrak{A}}_{\mathcal{V}}$ is defined inductively as shown in Figure 3.2, where $Q\nu$ (resp., $Z\nu$) denotes the application of variable valuation $\mathcal{V}$ to query Q (resp., variables Z), and $ans(Q\nu, \mathcal{I})$ denotes the result of evaluating the (boolean) query $Q\nu$ over the instance $\mathcal{I}$. Moreover, $\mathcal{I} \Rightarrow_{\mathfrak{A}} \mathcal{I}'$ holds iff the progression mechanism allows to progress from $\mathcal{I}$ to $\mathcal{I}'$. We say that a closed $\mu\mathcal{L}$ formula Φ holds for $\mathfrak{A}$, denoted as $\mathfrak{A} \models \Phi$, iff $\mathcal{I}_0 \in \Phi^{\mathfrak{A}}$. We call *model checking* verifying whether $\mathfrak{A} \models \Phi$ holds.

For the intuitions behind the semantics of μ-calculus we refer to Section 2.3. Here we focus on the differences between the language used for conjunctive artifacts and the one defined here for relational artifacts by comparing the semantics in Figure 2.1 and Figure 3.2. The major distinction is that conjunctive queries are evaluated under the certain answers semantics, while FO queries are evaluated following the usual semantics. Moreover, here we allow for quantification *across-state*, meaning that fixpoint operators may occurs in the scope of existential or universal quantifiers. However the scope of quantifiers is the set $\mathcal{C}_{\mathcal{I}_0}$ that, being finite, give rise to no technical issue.

Example 3.3.1 (Continues from Example 3.1.1)**.** *Now that we have defined our constraints formalism, we are in the position to express the constraints informally discussed in Example 3.1.1.*

For ReqOrders*, we first define formulas corresponding to the phases of the diagram in Figure 3.1(a):*

$$\begin{array}{lcl}
\psi_A & = & \neg\exists x,y,z.\mathsf{ROItem}(x,y,z) \\
\psi_B & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow z = \mathsf{requested}) \wedge \exists x,y.\mathsf{ROItem}(x,y,\mathsf{requested}) \\
\psi_C & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow (z = \mathsf{requested} \vee z = \mathsf{purchased})) \wedge \\
 & & \exists x,y.\mathsf{ROItem}(x,y,\mathsf{requested}) \wedge \exists x,y.\mathsf{ROItem}(x,y,\mathsf{purchased}) \\
\psi_D & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow z = \mathsf{purchased}) \wedge \exists x,y.\mathsf{ROItem}(x,y,\mathsf{purchased}) \\
\psi_E & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow (z = \mathsf{purchased} \vee z = \mathsf{shipped})) \wedge \\
 & & \exists x,y.\mathsf{ROItem}(x,y,\mathsf{purchased}) \wedge \exists x,y.\mathsf{ROItem}(x,y,\mathsf{shipped}) \\
\psi_F & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow z = \mathsf{shipped}) \wedge \exists x,y.\mathsf{ROItem}(x,y,\mathsf{shipped}).
\end{array}$$

Then, the dynamic constraints of ReqOrders *are captured by the formula*

$$\Phi_{RO} = \psi_A \wedge \nu Z.(\textstyle\bigwedge_{i=1,\ldots,11} \Phi_i \wedge \Box Z).$$

It requires that in the initial state of $\mathfrak{A}$ *there are no items included in any pending order, i.e., the relation* ROItem *is empty, and that all formulas* Φ_i *listed below hold in every state. Each of* Φ_1 *to* Φ_6 *corresponds to a single transition as in Figure 3.1(a), expressing the constraint that the artifact remains in its current phase* until *it reaches the following one, also requiring that such a phase is eventually reached in a finite number of steps, and that no other phase is reached until then:*

$$\begin{array}{lcllcl}
\Phi_1 & = & \psi_A \rightarrow \mu Z.(\psi_B \vee (\psi_A \wedge \Box Z)) & \Phi_4 & = & \psi_D \rightarrow \mu Z.(\psi_E \vee (\psi_D \wedge \Box Z)) \\
\Phi_2 & = & \psi_B \rightarrow \mu Z.(\psi_C \vee (\psi_B \wedge \Box Z)) & \Phi_5 & = & \psi_E \rightarrow \mu Z.(\psi_F \vee (\psi_E \wedge \Box Z)) \\
\Phi_3 & = & \psi_C \rightarrow \mu Z.(\psi_D \vee (\psi_C \wedge \Box Z)) & \Phi_6 & = & \psi_F \rightarrow \mu Z.(\psi_A \vee (\psi_F \wedge \Box Z)).
\end{array}$$

The remaining formulas express static constraints, specifically inclusion dependencies and range restrictions:

$$\begin{array}{lcl}
\Phi_7 & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow \exists u,v.\mathsf{RO}(x,u,v)) \\
\Phi_8 & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow \mathsf{Status}(z)) \\
\Phi_9 & = & \forall x.(\mathsf{Status}(x) \rightarrow (x = \mathsf{requested} \vee x = \mathsf{purchased} \vee x = \mathsf{shipped})) \\
\Phi_{10} & = & \forall x,y,z.(\mathsf{ROItem}(x,y,z) \rightarrow \exists w \mathsf{LineItem}(y,w)) \\
\Phi_{11} & = & \forall x,y,z.(\mathsf{RO}(x,y,z) \rightarrow (\mathsf{Requester}(y) \wedge \mathsf{Buyer}(z)).
\end{array}$$

For ProcOrders*, we just need to express some static specifications over instances. Hence,* Φ_{PO} *is the conjunction of the following formulas, expressing inclusion dependency constraints:*

$$\begin{array}{l}
\nu Z.(\forall x,y,z.(\mathsf{PO}(x,y,z) \rightarrow \mathsf{Supplier}(z)) \wedge \Box Z) \\
\nu Z.(\forall x,y,z.(\mathsf{POItem}(x,y,z) \rightarrow \exists u.\mathsf{PO}(x,y,u)) \wedge \Box Z) \\
\nu Z.(\forall x,y,z.(\mathsf{POItem}(x,y,z) \rightarrow \exists u.\mathsf{LineItem}(u,z)) \wedge \Box Z).
\end{array}$$

Finally, the set of inter-artifact dynamic constraints Φ_{inter} *is the conjunction of the following formulas:*

$$\begin{array}{l}
\nu Z.(\forall x,y.(\mathsf{ReqOrders.LineItem}(x,y) \leftrightarrow \mathsf{ProcOrders.LineItem}(x,y)) \wedge \Box Z) \\
\nu Z.(\forall x,y,z.(\mathsf{POItem}(x,y,z) \rightarrow \mathsf{ROItem}(y,z,\mathsf{purchased})) \wedge \Box Z) \\
\nu Z.(\forall x,y,z.(\mathsf{PO}(x,y,z) \rightarrow \exists w,k.\mathsf{RO}(y,w,k)) \wedge \Box Z).
\end{array}$$

The first formula requires that the LineItem *relations in both artifacts have the same set of tuples, the second one that every item belonging to a procurement order is also included in some requisition order, and the third one that every procurement order corresponds to a requisition order.* ■

We are interested in formally verifying properties of processes over artifact-based systems, in particular we are interested in *conformance* and *verification*, defined as follows:

Conformance. Given a process Π and an artifact system $\mathcal{A}$, the process is said to be acceptable if it fulfills all intra-artifact and inter-artifact dynamic constraints. In this case, we say that Π *conforms to* $\mathcal{A}$. In order to formally check *conformance*, we can resort to model checking and verify that:

$$\Upsilon(\Pi, \mathcal{A}) \models \Phi_{inter} \wedge \bigwedge_{i=1,\ldots,n} \Phi_i.$$

Verification. Apart from intra-artifacts and inter-artifact dynamic constraints, we are interested in other dynamic properties of the process over the artifact system. We say that a process Π over an artifact system $\mathcal{A}$ *verifies* a dynamic property Φ expressed in $\mu\mathcal{L}$ if

$$\Upsilon(\Pi, \mathcal{A}) \models \Phi.$$

It becomes evident that model checking of the transition system $\Upsilon(\Pi, \mathcal{A})$ generated by a process over an artifact system is the critical form of reasoning needed in our framework. We are going to study such a reasoning task next.

Example 3.3.2 (Continues from Example 3.3.1)**.** *We consider a process* $\Pi = \langle \boldsymbol{\rho}, \boldsymbol{\pi} \rangle$ *constituted by the following actions* $\boldsymbol{\rho}$ *and conditions-action rules* $\boldsymbol{\pi}$*. When specifying an action, we will use* $[\ldots]$ *to delimit each of the two parts* q_i^+ *and* Q_i^- *of the formula* $q_i^+ \wedge Q_i^-$ *in the left-hand side of an effect specification. Note that in such a formula the part corresponding to* Q_i^- *might be missing.*

Actions. *The set* $\vec{\rho}$ *of actions is the following. Action* $\mathsf{RequestItem}(r, i, b)$ *is used by the requester* r *to request a new line item* i *to buyer* b*. Such an action results in adding* i *to the requisition order of* r*. Notice that the RoCode denoting the requisition order is computed as a function of* r *and* b *only: performing this action multiple times for the same requester and buyer will result into adding line items to the same requisition order.*

$$\begin{aligned}\mathsf{RequestItem}(r, i, b) : \ \{\ & [\exists w.(\mathsf{Requester}(r) \wedge \mathsf{LineItem}(i, w) \wedge \mathsf{Buyer}(b))] \rightsquigarrow \\ & \{\mathsf{RO}(f(r, b), r, b),\ \mathsf{ROItem}(f(r, b), i, \mathsf{requested})\}, \\ & \mathit{CopyAll}\ \}\end{aligned}$$

Action $\mathsf{Purchase}(r, i, b, s)$ *is used by buyer* b *for purchasing an item* i *belonging to requisition order* r *from supplier* s*, thus creating (or updating) procurement orders (i.e., the relation* ProcOrders.PO*) and updating the status of the corresponding items kept by the relation* ReqOrders.ROItem*. Again, notice that PoCode is not a function of the item* i *passed as parameter. By writing* $\mathit{CopyAll} \setminus \mathsf{ROItem}$ *we denote the copy of all relations except* ROItem*.*

$$\begin{aligned}\mathsf{Purchase}(r, i, b, s) : \ \{\ & [\exists w.\mathsf{RO}(r, w, b) \wedge \mathsf{ROItem}(r, i, \mathsf{requested}) \wedge \mathsf{Supplier}(s)] \rightsquigarrow \\ & \{\mathsf{PO}(g(r, b, s), r, s),\ \mathsf{POItem}(g(r, b, s), r, i), \\ & \ \mathsf{ROItem}(r, i, \mathsf{purchased})\}, \\ & [\mathsf{ROItem}(x, y, z)] \wedge [\neg \mathsf{ROItem}(r, i, \mathsf{requested})] \rightsquigarrow \\ & \{\mathsf{ROItem}(x, y, z)\}, \\ & \mathit{CopyAll} \setminus \mathsf{ROItem}\ \}\end{aligned}$$

The following actions are used to ship all items included in a given procurement order p, and to deliver items belonging to a requisition order r to the corresponding requester, respectively. Notice that the first avoids copying all facts concerning p whereas the latter does the same with all facts related to r.

$$\begin{array}{ll} \mathsf{Ship}(p): & \{\ [\mathsf{POItem}(p,x,y) \wedge \exists z.\mathsf{ROItem}(x,y,z)] \rightsquigarrow \{\mathsf{ROItem}(x,y,\mathsf{shipped})\}, \\ & \ [\mathsf{POItem}(x,y,z)] \wedge [\neg\mathsf{POItem}(p,y,z)] \rightsquigarrow \{\mathsf{POItem}(x,y,z)\}, \\ & \ [\mathsf{PO}(x,y,z)] \wedge [\neg\mathsf{PO}(p,y,z)] \rightsquigarrow \{\mathsf{PO}(x,y,z)\}, \\ & \ \mathit{CopyAll} \setminus (\mathsf{POItem}\ \mathit{and}\ \mathsf{PO})\ \} \\ \mathsf{Deliver}(r): & \{\ [\mathsf{ROItem}(x,y,z)] \wedge [\neg\mathsf{ROItem}(r,y,z)] \rightsquigarrow \{\mathsf{ROItem}(x,y,z)\}, \\ & \ [\mathsf{RO}(x,y,z)] \wedge [\neg\mathsf{RO}(r,y,z)] \rightsquigarrow \{\mathsf{RO}(x,y,z)\}, \\ & \ \mathit{CopyAll} \setminus (\mathsf{ROItem}\ \mathit{and}\ \mathsf{RO})\ \} \end{array}$$

Condition-action rules. *In each condition-action rule of our process, we instantiate the parameters passed to the action, while simply checking that they are meaningful, i.e., that they are in the current instance. Hence:*

$$\begin{array}{ll} \boldsymbol{\pi} = \{ & \exists x.(\mathsf{Requester}(r) \wedge \mathsf{LineItem}(i,x) \wedge \mathsf{Buyer}(b)) \mapsto \mathsf{RequestItem}(r,i,b), \\ & \exists x.(\mathsf{RO}(r,x,b) \wedge \mathsf{ROItem}(r,i,\mathsf{requested}) \wedge \mathsf{Supplier}(s)) \mapsto \mathsf{Purchase}(r,i,b,s), \\ & \exists x,y.\mathsf{PO}(p,x,y) \mapsto \mathsf{Ship}(p), \\ & \exists x,y.\mathsf{RO}(r,x,y) \mapsto \mathsf{Deliver}(r)\ \} \end{array}$$

We close our example by observing that the process we have specified conforms to the lifecycle in Example 3.3.1. ■

3.4 Decidability

We now address the problem of how to compute conformance and verification over relational artifact systems. We first show that they are both undecidable in general, even in simple cases, and then we present a class of artifact systems for which verification is decidable. The undecidability result does not come as a surprise, since the transition system of a process over an artifact system can easily be infinite-state. Moreover, our framework is so general that it does not force the infinite state space regularity usually needed to apply known results on model checking on infinite state systems. However, we show that the undecidability holds even in a very simple case.

We consider a relational artifact system of the form $\mathcal{A}_u = \langle\{A\}, \mathtt{true}\rangle$ with $A = \langle\vec{R}, I_0, \mathtt{true}\rangle$. That is $\mathcal{A}_u$ is formed by a single artifact A with no intra-artifact or inter-artifact dynamic constraints. In addition, we consider processes with only one action ρ_u and only one condition-action rule $\mathtt{true} \mapsto \rho_u$ that has a $\mathtt{true}$ condition and hence allows the execution of the action ρ_u at every moment. The action ρ_u is without parameters, its effects have the form

$$q_i^+ \rightsquigarrow I_i',$$

where q_i^+ is a CQ (hence without any form of negation and of universal quantification), and it includes *CopyAll* effects. We call these kinds of relational artifact systems and processes *simple*. Next lemma shows that it is undecidable to verify in such cases the $\mu\mathcal{L}$ formula $\mu Z.(q \vee \langle - \rangle Z)$, expressing that there exists a sequence of action executions that leads to an instance where a boolean CQ q holds.

Lemma 3.1. *Verifying whether the $\mu\mathcal{L}$ formula $\mu Z.(q \vee \langle - \rangle Z)$ holds for a simple process over a simple artifact is undecidable.*

Proof. We observe that we can use the set of effects of ρ_u to encode a set of tuple-generating dependencies (TGDs) [2]. Hence we can reduce to the above verification problem the problem of answering boolean CQs in a relational database under a set of TGDs, which is undecidable [15] (in fact, special care is needed because of the use of Skolem terms instead of labeled nulls.) □

Theorem 3.1. *Conformance checking and verification are both undecidable for processes over relational artifacts systems.*

Proof. Lemma 3.1 gives us undecidability of verification, already for simple relational artifact systems and processes. To get undecidability of conformance it is sufficient to consider the simple process Π_u over relational artifact systems of the form $\mathcal{A}_{cu} = \langle \{A_c\}, \texttt{true} \rangle$, with $A_c = \langle \vec{R}, I_0, \mu Z.(q \vee \langle - \rangle Z) \rangle$. Note that $\mathcal{A}_{cu}$ is a variant of simple artifact systems $\mathcal{A}_u$ in which the artifact has as intra-artifact dynamic constraint exactly $\mu Z.(q \vee \langle - \rangle Z)$. The claim follows again from Lemma 3.1, considering that, by definition, checking conformance of the simple process Π_u wrt $\mathcal{A}_{cu}$ is equivalent to checking whether $\Upsilon(\Pi_u, \mathcal{A}_u) \models \mu Z.(q \vee \langle - \rangle Z)$. □

Next we tackle decidability, and, following the line of the conjunctive setting is Chapter 2, we adapt recent results on data exchange [87] by isolating a notable case of processes over relational artifact systems for which both conformance and verification are decidable. Our results rely on the possibility of building a special process that we call "positive approximate". For such a process there exists a tight correspondence between the application of an action and a step in the chase of a set of TGDs [2, 87].

Given a process $\Pi = \langle \boldsymbol{\rho}, \boldsymbol{\pi} \rangle$, the *positive approximate* of Π is the process $\Pi^+ = \langle \boldsymbol{\rho}^+, \boldsymbol{\pi}^+ \rangle$ obtained from Π as follows. For each action ρ in $\boldsymbol{\rho}$, there is an action ρ^+ in $\boldsymbol{\rho}^+$, obtained from ρ by

- removing all input parameters from the signature, and
- substituting each effect $q_i^+ \wedge Q_i^- \rightsquigarrow I_i'$ with the one that uses only the *positive* part of the head of the effect specification, i.e., with $q_i^+ \rightsquigarrow I_i'$.

Note that the variables in q_i^+ that used to be parameters in ρ, become free variables in ρ^+. Then, for each condition-action rule $Q \mapsto \rho$ in $\boldsymbol{\pi}$, there is a rule $\texttt{true} \mapsto \rho^+$ in $\boldsymbol{\pi}^+$. Hence, Π^+ allows for executing every action at every step.

Now, relying again on the parallelism between chase in data exchange and action execution in artifact systems, we take advantage of the notion of weak acyclicity in data exchange [87] to devise an interesting class of processes which are guaranteed to generate a finite-state transition system, when run over a relational artifact system. This in turn guarantees decidability of conformance and verification.

Let Π be a process over an artifact system $\mathcal{A}$, and $\Pi^+ = \langle \boldsymbol{\rho}^+, \boldsymbol{\pi}^+ \rangle$ its positive approximate. We call *dependency graph* of Π^+ the following (edge labeled) directed graph:

Nodes: for every artifact $A = \langle \boldsymbol{R}, I_0, \Phi \rangle$ of $\mathcal{A}$, every relation symbol $R_i \in \boldsymbol{R}$, and every attribute att or R_i, there is a node (R_i, att) representing a position;

Edges: for every action ρ^+ of $\boldsymbol{\rho}^+$, every effect $q_i^+(\boldsymbol{t}) \rightsquigarrow I_i'(\boldsymbol{t}', f_1(\boldsymbol{t}_1), \ldots, f_n(\boldsymbol{t}_n))$ of ρ^+ (where for convenience we have made explicit the terms occurring in q_i^+ and I_i', and where consequently $\boldsymbol{t}', \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n \subseteq \boldsymbol{t}$ are either constants or variables), every variable $x \in \boldsymbol{t}$, and every occurrence of x in q_i^+ in position p, there are the following edges:

- for every occurrence of x in I_i' in position p', there is an edge $p \to p'$;
- for every Skolem term $f_k(t_k)$ such that $x \in \boldsymbol{t}_k$ occurs in I_i' in position p'', there is a *special edge* (i.e., one labeled by $*$) $p \xrightarrow{*} p''$.

We say that Π is *weakly acyclic* if the dependency graph of Π^+ has no cycle going through a special edge.

Intuitively, ordinary edges keep track of the fact that a value may propagate from position p to position p' in a possible trace. Moreover, special edges keep track of the fact that a value in position p can be taken as parameter of a Skolem function, thus contributing to the creation of a (not necessarily new) value in any position p''. If a cycle goes through a special edge, then a new value appearing in a certain position may determine the creation of another one, in the same position, later during the execution of actions. Since this may happen again and again, no bound can be put on the number of newly generated Skolem terms, and thus on the number of new values appearing in the instance. Note that the definition allows for cycles as long as they do not include special edges.

Lemma 3.2. *Let* Π *be a* weakly acyclic *process over a relational artifact system* $\mathcal{A}$ *with initial instance* $\mathcal{I}_0$*, and let* Π^+ *be the positive approximate of* Π*. Then there exists a polinomial in the size of* $\mathcal{I}_0$ *that bounds the size of every instance generated by* Π^+*.*

Proof. For every node p in the dependency graph of Π^+ we define an incoming path to be any (finite or infinite) path ending in p. We define the rank of p, denoted by $\underline{\text{rank}}(p)$, as the maximum number of special edges on any such incoming path. Since Π^+ is weakly acyclic, there are no cycles going through special edges. Thus $\underline{\text{rank}}(p)$ is finite. Let r be the maximum, among $\underline{\text{rank}}(p_i)$, over all positions and let s be the total number of positions p in the schema (equal to the number of nodes in the graph). The latter number is a constant, since the schema is fixed. Notice that r is at most s, since we cannot have a path going to the same node twice using special edges, otherwise the graph would contain a cycle going through special edges. The next observation is that we can partition the nodes in the dependency graph, according to their rank, into subsets $\{N_0, N_1, \ldots, N_r\}$ where N_i is the set of all nodes with rank i. Let n be the total number of distinct values that occur in the instance $\mathcal{I}_0$. Let $\mathcal{I}'$ be any instance obtained from $\mathcal{I}_0$ after some arbitrary actions sequence. We prove by induction on i the following claim: for every i there exists a polynomial P_i such that the total number of distinct values that occur in $\mathcal{I}'$ at positions in N_i is at most $P_i(n)$.

Base case: If p is a position in N_0, then there are no incoming paths with special edges. Thus no new values (i.e., Skolem terms) are ever created at position p during

the execution of actions. Hence, the values occurring in $\mathcal{I}'$ at position p are among the n values of the original instance $\mathcal{I}_0$. Since this is true for all the positions in N_0, we can then take $P_0(n) = n$.

Inductive case: The first kind of values that may occur in $\mathcal{I}'$ at a position in N_i are those values that already occur in $\mathcal{I}_0$ at the same position. The number of such values is at most n. In addition, a value may occur in $\mathcal{I}'$ at a position in N_i for two reasons: by being copied from some position in N_j with $j \neq i$, as a result of an action, or by being generated as a possibly new skolem term, also as a result of an action. We count first how many skolem terms can be generated at most. Let p be some position in N_i. A new value can be generated in p as a result of an action only due to special edges. But any special edge that may enter p must start at a node in $N_0 \cup \ldots \cup N_{i-1}$. Applying the inductive hypothesis, the number of distinct values that can exist in all the nodes in $N_0 \cup \ldots \cup N_{i-1}$ is bounded by $H(n) = P_0(n) + \ldots + P_{i-1}(n)$. Let d be the maximum number of special edges that enter a position, over all positions in the schema. Then for every choice of d-tuples of values in $N_0 \cup \ldots \cup N_{i-1}$ (one component for each special edge that can enter a position) and for every action in Π^+, the number of new values generated at position p is bounded by $c_{\Pi^+} \cdot H(n)^d$ where c_{Π^+} is the size of Π^+ seen as a string. Notice that this number does not depend on the data in $\mathcal{I}_0$. (In particular we get close to the number $c_{\Pi^+} \cdot H(n)^d$ of new values when actions are using the same relation over and over with different Skolem functions, all of the maximum (d) arity.) If we consider all positions p in N_i, the total number of values that can be generated is bounded by $|N_i| \cdot c_{\Pi^+} \cdot H(n)^d$ where $|N_i|$ is the number of positions in N_i. Let $G(n) = |N_i| \cdot c_{\Pi^+} \cdot H(n)^d$. Obviously, G is a polynomial. We count next the number of distinct values that can be copied to positions of N_i from positions of N_j with $j \neq i$. Such copying can happen only if there are non-special edges from positions in N_j with $j \neq i$ to positions in N_i . We observe first that such non-special edges can originate only at nodes in $N_0 \cup \ldots \cup N_{i-1}$, that is, they cannot originate at nodes in N_j with $j > i$. Otherwise, assume that there exists $j > i$ and there exists a nonspecial edge from some position in N_j to a position p of N_i . Then the rank of p would be larger than i, which is a contradiction. Hence, the number of distinct values that can be copied in positions of N_i is bounded by the total number of values in $N_0 \cup \ldots \cup N_{i-1}$, which is $H(n)$ from our previous consideration. Putting it all together, we can take $P_i(n) = n + G(n) + H(n)$. Since P_i is a polynomial, the claim is proven. In the above claim, i is bounded by the maximum rank r, which is a constant. Hence, there exists a fixed polynomial P such that the number of distinct values that can exist in $\mathcal{I}'$, over all positions, is bounded by $P(n)$. In particular, the number of distinct values that can exist in $\mathcal{I}'$ at a single position is also bounded by $P(n)$. Then the total number of tuples in $\mathcal{I}'$ is bounded by $P(n)^s$ (recall that s is the number of all positions in the schema). It follows that the total number of facts in $\mathcal{I}'$ that can exist over all an instance is at most $\ell \times P(n)^s$, where ℓ is the number of relations in the schema. This is a polynomial in n since ℓ and s are constants being the schema fixed. Hence we get the thesis. □

Notice that as a direct result of this lemma, the transition system generated by the positive approximate over $\mathcal{A}$ has a number of states that is finite, and in fact at most exponential in the size of the initial instance $\mathcal{I}_0$ of $\mathcal{A}$. Now we show that a similar result holds for the original process Π. The key to this is the following observation that easily follows from the definition of ρ^+ for an action ρ.

Lemma 3.3. *For every action ρ over $\mathcal{A}$, instances $\mathcal{I}_1$, $\mathcal{I}_2$ of $\mathcal{A}$, and ground substitution σ for the parameters of ρ, if $\mathcal{I}_1 \subseteq \mathcal{I}_2$ then $do(\rho\sigma, \mathcal{I}_1) \subseteq do(\rho^+, \mathcal{I}_2)$.*

We can extend the result above to any sequence of actions, by induction on the length of the sequence. Hence, we get that the instance obtained from the initial instance by executing a sequence of actions of the original process Π is contained in the instance obtained by executing the same sequence of actions of Π^+. From this observation, considering the bound in Lemma 3.2, we get the desired result for the original process.

Lemma 3.4. *Let Π be a* weakly acyclic *process over a relational artifact system $\mathcal{A}$ with initial instance $\mathcal{I}_0$. Then there exists a polinomial in the size of $\mathcal{I}_0$ that bounds the size of every instance generated by Π.*

From this, we obtain our main result.

Theorem 3.2. *Conformance and verification of $\mu\mathcal{L}$ formulas are decidable for weakly acyclic processes over relational artifact systems.*

Proof. From Lemma 3.4, it follows that the transition system generated by a weakly acyclic process over a relational artifact system $\mathcal{A}$ has a number of states that is at most exponential in the size of the initial instance $\mathcal{I}_0$ of $\mathcal{A}$. The claim then follows from known results on verification of μ-calculus formulas over finite transition systems (see e.g., [61]). □

From the exponential bound on the number of states of the generated transition system mentioned in the proof above, we get not only decidability of verification and conformance, but also an ExpTime upper bound for its computational complexity (assuming a bound on the nesting of fixpoints).

3.5 A variant: relational data-centric dynamic systems

Stemming from our work of relational artifact-centric systems, others have investigated a variant called relational data-centric dynamic systems (DCDS) [11]. Here we discuss this model that we are going to use later on when looking at concrete artifact-centric and data-aware models. Relational Data-Centric Dynamic Systems (DCDSs) have been strongly inspired by the relational and conjunctive artifact-centric frameworks, and indeed they kept the core elements we presented before.

A DCDSs is a couple $\mathcal{S} = (\mathcal{D}, \mathcal{P})$ formed by two layers: the *data layer* $\mathcal{D}$ used to hold the relevant information to be manipulated by the system and the *process layer* $\mathcal{P}$ formed by *actions* and a *process* based on them, that characterizes the dynamic behavior of the system.

The *data layer* is a tuple $\mathcal{D} = (\mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0)$ where:

- $\mathcal{C}$ is the countably infinite domain;
- $\mathcal{R} = \{R_1, \ldots, R_n\}$ is a database schema, constituted by a finite set of relation schemas;

- $\mathcal{E}$ is a finite set $\{\mathcal{E}_1, \ldots, \mathcal{E}_m\}$ of equality constraints. Each $\mathcal{E}_i$ has the form $Q_i \rightarrow \bigwedge_{j=1,\ldots,k} z_{ij} = y_{ij}$, where Q_i is a domain independent FO query over $\mathcal{R}$ using constants from the active domain of $\mathcal{I}_0$, denoted $\textsc{adom}(\mathcal{I}_0)$, whose free variables are $\vec{x}$, and z_{ij} and y_{ij} are either variables in $\vec{x}$ or constants in $\textsc{adom}(\mathcal{I}_0)$
- $\mathcal{I}_0$ is a database instance that represents the initial state of the data layer, which conforms to the schema $\mathcal{R}$ and *satisfies* the constraints $\mathcal{E}$: namely, for each constraint $Q_i \rightarrow \bigwedge_{j=1,\ldots,k} z_{ij}{=}y_{ij}$ and for each tuple (i.e., substitution for the free variables) $\theta \in ans(Q_i, \mathcal{I})$, it holds that $z_{ij}\theta = y_{ij}\theta$

As usual, $t\theta$ (resp., $\varphi\theta$) denotes the term (resp., the formula) obtained by applying the substitution θ to t (resp., φ) and $ans(Q, \mathcal{I})$ is the set of assignments θ from the free variables of the query Q to the domain of database instance $\mathcal{I}$, such that $\mathcal{I} \models Q\theta$. We treat $Q\theta$ as a boolean query, and with some abuse of notation, we say $ans(Q\theta, \mathcal{I}) \equiv true$ if and only if $\mathcal{I} \models Q\theta$.

The *process layer* over a data layer $\mathcal{D}$ is a tuple $\mathcal{P} = (\mathcal{F}, \mathcal{A}, \varrho)$ where:

- $\mathcal{F}$ is a finite set of *functions* each representing the interface to an *external service*. When the service is called, a result is activated and an answer is produced.
- $\mathcal{A}$ is a finite set of *actions* whose executions modify the data layer and may involve external service calls.
- ϱ is a finite set of *condition-action rules* which specifies which actions can be executed.

Notice that actions have the same form of those in Chapter 3, but now Skolem functions are now replaced with functions in $\mathcal{F}$. An *action* $\alpha \in \mathcal{A}$ has the form:

$$\alpha(p_1 \ldots p_n) : \{e_1 \ldots e_m\}$$

where, once more, $\alpha(p_1 \ldots p_n)$ is the *signature* of the action, α is the name and $p_1 \ldots p_n$ are the *input parameters* that need to be substituted with values for the execution of the action. All *effect specifications* $e_1 \ldots e_m$ take place simultaneously and each e_i has the form $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$ where:

- $q_i^+ \wedge Q_i^-$ is a query over $\mathcal{R}$ whose terms are variables, action parameters, and constants from $\textsc{adom}(\mathcal{I}_0)$, where q_i^+ is a UCQ, and Q_i^- is an arbitrary FO formula whose free variables are among those of q_i^+. Intuitively, q_i^+ selects the tuples to instantiate the effect with, and Q_i^- filters away some of them.
- E_i is the effect, i.e., a set of facts for $\mathcal{R}$, which includes as terms: terms in $\textsc{adom}(\mathcal{I}_0)$, free variables of q_i^+ and Q_i^- (including action parameters), and in addition Skolem terms formed by applying a function $f \in \mathcal{F}$ to one of the previous kinds of terms. Such Skolem terms involving functions represent external service calls and are interpreted so as to return a value chosen by an external user/environment when executing the action.

The *process* ϱ is a finite set of *condition-action rules*, of the form $Q \mapsto \alpha$, where α is an action in $\mathcal{A}$ and Q is a FO query over $\mathcal{R}$ whose free variables are exactly the parameters of α, and whose other terms can be either quantified variables or constants in $\textsc{adom}(\mathcal{I}_0)$.

Notice that from the syntactic viewpoint, a DCDS and a RAS are very similar. Differences rise in the semantics, that, given a DCDS $\mathcal{S}$, is defined, as usual, by means of the transition system $\Upsilon_{\mathcal{S}}$ representing the possible evolutions of $\mathcal{S}$.

In [11] two semantics are given. The first one assumes external service to behave *deterministically*. This means that whenever an external service $f \in \mathcal{F}$ is called twice with the same parameters, it must return the same values. The second one considers services to behave *nondeterministically*, i.e., two identical calls to the same service may return different results.

In this thesis, we are interested in the second scenario, as it is more general and captures the case of services that model human operators. In what follows we hence describe the nondeterministic service semantics, and we refer to [11] for the deterministic one.

Let $\mathcal{S} = (\mathcal{D}, \mathcal{P})$ a DCDS with data layer $\mathcal{D} = (\mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0)$ and process layer $\mathcal{P} = (\mathcal{F}, \mathcal{A}, \varrho)$. A *state* of the system is simply a relational instance of $\mathcal{R}$ over $\mathcal{C}$ satisfying constraints in $\mathcal{E}$. The initial state is $\mathcal{I}_0$.

We now define the semantics of action application. Let α be an action in $\mathcal{A}$ of the form $\alpha(p_1 \dots p_m) : \{e_1 \dots e_m\}$ with effects $e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E_i$. The parameters of α are guarded by the condition-action rule $Q \mapsto \alpha \in \varrho$. Let σ be a legal substitution for the input parameters $p_1 \dots p_m$ with values taken from $\mathcal{C}$. The effects of firing an action α with parameters σ over an instance $\mathcal{I}$ is defined as:

$$\textsc{do}(\mathcal{I}, \alpha\sigma) = \bigcup_{q_i^+ \wedge Q_i^- \rightsquigarrow E_i \in \{e_1 \dots e_m\}} \; \bigcup_{\theta \in ans((q_i^+ \wedge Q_i^-)\sigma, \mathcal{I})} E_i\sigma\theta$$

that coincides with the definition of *do* given in Chapter 3. Notice that DO generates an instance over values from the domain $\mathcal{C}$ but also over Skolem terms, which model service calls. For any such instance $\overline{\mathcal{I}}$, we denote with $\textsc{calls}(\overline{\mathcal{I}})$ the set of calls it contains. For a given set $D \subseteq \mathcal{C}$, we denote with $\textsc{evals}_D(\mathcal{I}, \alpha, \sigma)$ the set of substitutions that replace all service calls in $\textsc{do}(\mathcal{I}, \alpha, \sigma)$ with values in D,

$$\textsc{evals}_D(\mathcal{I}, \alpha, \sigma) = \{\theta \mid \theta \text{ is a total function } \theta : \textsc{calls}(\textsc{do}(\mathcal{I}, \alpha, \sigma)) \to D\}$$

Each substitution in $\textsc{evals}_D(\mathcal{I}, \alpha, \sigma)$ models the simultaneous evaluation of all service calls, which replaces the calls with results selected nondeterministically from D. In the following, we refer to these substitutions as *evaluations*.

Evaluations are what differentiate DCDSs from RASs. They indeed represents the actual possible inputs from the external values.

Next we define the transition relation $\textsc{n-exec}_{\mathcal{S}}$ between states, called *concrete execution* of $\alpha\sigma\theta$. The tuple $(\mathcal{I}, \alpha\sigma\theta) \in \textsc{n-exec}_{\mathcal{S}}$ if the following holds:

1. σ is a legal parameter assignment for α in a state $\mathcal{I}$;
2. $\theta \in \textsc{evals}_{\mathcal{C}}(\mathcal{I}, \alpha, \sigma)$;
3. $\mathcal{I}' = \textsc{do}(\mathcal{I}, \alpha, \sigma)\theta$ and

4. $\mathcal{I}'$ satisfies constraints in $\mathcal{E}$.

Notice that, even though services are nondeterministic, *within* a concrete execution step all occurrences of the same service call evaluate to the same result (modeling the fact that a call with a given arguments is invoked only once per transition, and the returned result is copied as needed).

We can now define the possibly infinite *concrete transition system* $\Upsilon_{\mathcal{S}}$ for $\mathcal{S}$. Formally, $\Upsilon_{\mathcal{S}} = (\mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Rightarrow)$ where:

- the initial state $s_0 = \mathcal{I}_0$;
- db is such that $db(\mathcal{I}) = \mathcal{I}$ and
- the set of states Σ and the transition relation $\Rightarrow$ are defined by simultaneous induction as the smallest sets satisfying the following properties:
  - $\mathcal{I}_0 \in \Sigma$;
  - if $\mathcal{I} \in \Sigma$, then for all α, σ, θ and $\mathcal{I}'$ such that $(\mathcal{I}, \alpha\sigma\theta, I') \in \text{N-EXEC}_{\mathcal{S}}$, we have $\mathcal{I}' \in \Sigma$ and $\mathcal{I} \Rightarrow \mathcal{I}'$.

It is easy to see that there is a strong relationship between the semantics of DCDSs and the semantics of RASs. Actions have indeed the same semantics in the two frameworks. The big difference lies in the interpretation of the functions, that here are associated to value of the domain (through $\text{EVALS}_{\mathcal{S}}$), while in RASs this mapping is missing.

We now turn to the verification formalism. DCDSs are clearly more expressive than RASs. It is enough to analyze the behavior of a DCDS $\mathcal{S}$ by observing its transition system $\Upsilon_{\mathcal{S}}$: at each step it shows an infinite branch if there is an action with a function in one of its effects. Achieving decidability of verification is indeed a challenge. The authors tackle the problem by both restricting the verification language and constraining the evolution of the system.

The fragment of first-order μ-calculus that is considered in [11] is called *persistence preserving mu-calculus*, denoted by $\mu\mathcal{L}_P$. It restricts the variant of μ-calculus presented in Chapter 3 by requiring that quantification spans over individuals that persists along the system evolution only. To enforce such a restriction, a special predicate $\text{LIVE}(x)$ which states that x belongs to the current active domain is used. With a slight abuse of notation, in the following we write $\text{LIVE}(x_1 \dots x_n) \equiv \text{LIVE}(\vec{x}) \equiv \bigwedge_{i \in \{1 \dots n\}} \text{LIVE}(x_i)$.

The language $\mu\mathcal{L}_P$ is defined as follows:

$$\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}(x) \wedge \Phi \mid \langle - \rangle(\text{LIVE}(\vec{x}) \wedge \Phi) \mid [-](\text{LIVE}(\vec{x}) \wedge \Phi) \mid Z \mid \mu Z.\Phi$$

where Q is a possibly open FO query, Z is a second-order predicate variable, and the following assumption holds: in $\langle - \rangle(\text{LIVE}(\vec{x}) \wedge \Phi)$ and in $[-](\text{LIVE}(\vec{x}) \wedge \Phi)$, the variables in $\vec{x}$ are exactly the free variables of Φ.

The use of predicate $\text{LIVE}(\cdot)$ in $\mu\mathcal{L}_P$ ensures that individuals are only considered if they persist along the system evolution, while the evaluation of a formula with individuals that are not present in the current database trivially leads to false or true (depending on the use of negation).

Concerning the restriction on the process, the authors call a DCDS $\mathcal{S}$ *state bounded* if there is a finite bound b such that for each state $\mathcal{I}$ of $\Upsilon_{\mathcal{S}}$, $|\textsc{adom}(\mathcal{I})| < b$. Notice that state boundedness allows runs in which infinitely many distinct values occur: they can be distributed *across* states of the run but not *within* a single state.

The main results in [11] states that *verification of $\mu\mathcal{L}_P$ properties by state-bounded DCDS with nondeterministic services is decidable*. To get such a result, a suitable form of isomorphism and bisimulation is used to finitely abstract the execution transition system. The main intuition is that, even though infinitely many different values can appear during the system, the language discerns only the finite subset of those that persist.

State boundedness is a semantic property and it turn out to be undecidable. The authors hence propose a checkable syntactic property that guarantees state-boundedness, called GR-aciclicity. Such a notion is less restrictive than the (classic) acyclicity presented in Chapter 2 and 3, because here we do allow infinitely many new values in a run. The behavior we would like to avoid is the following. By definition of the DCDS action semantics, a transition "forgets" all values that are not explicitly recalled. Therefore to accumulate infinitely many values in a state it is necessary that an effect generates new values in a position that is always recalled (by being copied). The GR-acyclicity basically requires the absence of a "generate cycle" that feeds into a "recall state".

3.6 Discussion

We presented a powerful framework for modeling artifact-centric business processes. Its power relies in the expressivity of actions, that, unlike the original framework in Chapter 2, allows for negation. We actually have a *guarded* form of negation because, given an infinite domain, we should guarantee query answering to return a finite set of tuples. For this reason, in an effect $q_i^+ \wedge Q_i^- \rightsquigarrow I_i'$, free variables appearing in Q_i^- have also to be included in q_i^+, that is a conjunctive query.

Having negation in actions comes with the cost of giving up incomplete information. Indeed when adding negation, the whole theory of conjunctive queries collapses.

Even though the relational settings appear to be more appealing from a practical viewpoint, the theory behind the conjunctive artifact-centric systems in Chapter 2 is more convoluted, and it has been thought for scenarios in artificial intelligence. If we think, e.g., of our database as the knowledge base of an agent, the assumption of complete knowledge is too strong. The relational framework, however, fits much better the context of business processes and, in order to prove that, in the next Chapters we will see how DCDSs (that has been strongly inspired by the relational setting presented in this chapter) are able to simulate the execution of other formalisms that are used to model data-aware business processes.

One of the main characteristic of our approach is to abstract external input from the user as Skolem functions. However, we left in purpose Skolem terms as "handlers" for values in the domain in order to be abstract enough to permit different semantics interpretations to them. Conversely, to adapt our framework to describe actual scenarios, we need to associate values to Skolem functions. To this aim,

we summarized in Section 3.5 the main concepts and results presented in [11]. The main differences with the work presented here are: *(i)* Skolem functions are mapped to domain values and *(ii)* the data layer is equipped with equality constraints. Interpreting Skolem functions leads to infinitely branching. Hence, bounding the number of different values within a run does not guarantee decidability: it is also necessary to abstract from the possible interpretation values (using bisimulation as in Chapter 2).

There is an important difference between DCDS and the work in [57, 46] discussed in Section 1.2. In [57, 46] properties of interest are checked for every possible instance, while in our work and in DCDSs there is a fixed initial instance. However, their formalism is less expressive than μ-calculus, in what a linear time analysis only is allowed.

Besides [11], other works has been inspired from the framework presented in this Chapter, and, hence by those in Chapter 2. In [17] actions are modeled as pre- and post-conditions expressed in first-order logics and interpreted over the active domain. Actions parameters may introduce new values in the systems, and hence the number of states of the system is infinite in general. Inspired by work on systems abstraction [42, 41], when a condition (called b-boundedness) on the number of values in the active domain of system's states is imposed, the system can be abstracted as a finite transition system that can be therefore model-checked using standard techniques. To verify a first-order existential CTL property on the original system the (sound but not complete) algorithm consists in iteratively increasing the bound b until the abstract system verifies the property: this entails that the property holds in the concrete transition system as well.

The work [16, 18] extends [17] in what the first-order temporal specification language is augmented with epistemic operators. The language in [18] is the more expressive of the three because it also allows for alternation of quantifier and temporal operators and hence the technical development in [17], and in particular the notion of bisimulation, is extended and refined to cope with such a language.

Next we discuss the connection of our framework and our results with the field of reasoning about actions, starting with situation calculus.

Situation calculus tackles the hard task of formalizing the real world, and indeed it faces the typical issues of large scenarios, such as the ramification, qualification and frame problem. Our framework has instead been thought to model business processes which, by themselves, not only represent a small fraction of the world, but also abstractions and simplifications are often put in place. As the setting of business processes is more restricted than the real world, different assumptions can be made, and hence the aforementioned problems are less significant in practice, even though we consider them relevant in broader settings, and a further study on them is desirable and left as future work. However, our progression mechanism is still general and complex enough to expose all difficulties we need to overcome in order to get decidability of verification.

The second difference relies in how the two logical frameworks are specified: situation calculus is an axiomatic system where situations (or time instants) are explicitly represented, while our framework is a way for describing how artifacts (or more generally objects) evolve.

Also, situation calculus actions are *deterministic*, meaning that firing an action in the same configuration will result in the same effects. This is not the case of our framework in general and DCDSs in particular, where the result of an action can have different outcomes depending on the values we associate to the newly introduced functions.

Having said that, when we restrict to the case of complete information and we do not allow to use situations i.e., histories, in the verification formalism, then DCDSs can "simulate" situation calculus. More precisely, we can associate situation calculus fluents and predicates with relational predicates in a DCDS and, given a set of situation calculus axioms, we can build a DCDS which execution transition system generates all and only instances representing possible situation calculus theorems. Notably, not only DCDSs can simulate situation calculus, but it is also possible to carry the notion of state boundedness from DCDSs to situation calculus. Bounded situation calculus theories are shown in [53, 54], which, starting from the notion of state boundedness in DCDSs, bound the number of ground fluents in all situations by a constant. Under such a restriction, bounded situation calculus theories allow for verification of (a restricted form of) μ-calculus, although they have an infinite domain and an infinite set of states.

Our framework presents several similarities also with STRIPS. First of all, they both provide a progression mechanism rather than a logical axiomatization; they both provide no explicit solution to the frame problem and finally they do not represent histories and actions in the snapshots. Semantics of actions is instead quite different, since, in a way, ours provide only the list of facts to be added, given that each fact that is intended to be maintained has to be explicitly copied. Conversely, STRIPS next snapshot is building from the old one by first removing the delete list and then adding the add list facts. Moreover, we have conjunctive queries in action preconditions, hence we can use existential variables, instead in STRIPS only set of facts are allowed. Considering the semantics of the two formalisms, both the aforementioned syntactic dissimilarities does not result in a semantic gap. What does really differentiate relational artifact-centric systems from STRIPS is the ability to add fresh values in databases. In STRIPS the domain is defined in the planning problem, and it is indeed finite.

Chapter 4

The Guard-Stage-Milestone concrete artifact model

In this Chapter we present a concrete model for business artifact lifecycles, called *Guard-Stage-Milestone* (GSM). We then show how it is possible to translate a GSM schema into a DCDS schema. The purpose of the translation is twofold: on the one side, we show that the core elements of the framework presented in Chapter 3 are expressive enough to capture concrete models and, on the other side, the translation gives us a procedure to exploit results in [11] for verification purposes.

GSM has been created by IBM [82, 47] with the aim of incorporating business-level constructs in an intuitive manner, in order to be easy to use by business stakeholders. The motivation for the choice of GSM are several. First of all it is founded on the idea of artifact, the core notion that inspired our work. Indeed, concepts such as *information model* and *lifecycle*, that are dominant in this thesis, has been carried from GSM and previous works by IBM [109, 81, 23]. Even though we focus on verification while GSM has been thought to address more practical data-aware modeling difficulties, a comparison between GSM and our work is imperative. Besides, GSM has a precise semantics that provides the formal basis for both implementation and mathematical investigation. Lastly, its main constructs have been recently adopted by the Object Management Group (OMG) as the standard for Case Management Mondeling Notation (CMMN) [112].

There are four key elements in the GSM model: the *information model* for business artifacts, as in all variations of the artifact paradigm; *milestones*, which correspond to business-relevant operational objectives; *stages*, which correspond to clusters of activities intended to achieve milestones; and *guards*, which control when stages are activated and, as with milestones, are controlled through triggering events and/or conditions. Parallelism of activities is obtained by allowing multiple stages to be open at the same time, while a hierarchical structure of stages, support modularity.

The main feature of the GSM model is that, being declarative, it provides a flexible and goal-driven approach for modeling business processes.

The operational semantics for GSM is specified in terms of how a single *incoming event* is incorporated into the current status of the system, called *snapshot*, that is a description of all relevant aspects at a given moment of time. The “incorporation”

of an event means, roughly speaking, to process all the effects the event itself fires in the system. Such effects are determined based on a set of Event-Condition-Action (ECA) rules. A business step, or *B-step*, is exactly the transition from a snapshot of the system (before processing the event) to a new one, resulting from the incorporation of the event. The kind of effects the processing of an event fires are, informally, which milestones are achieved or invalidated, which stages become active or inactive and eventually, modification to the information model of artifacts. Of course, changes in milestone and/or stage status can trigger further status changes in the snapshot, but a B-step corresponds to the smallest unit of business-relevant change that can occur to a GSM system.

IBM also implemented a prototype, called Barcelona, to support experiments of both modeling and execution of a GSM schema. Indeed, the incremental semantics introduced in [82, 47] and reported here, serves as a basis for implementation of Barcelona GSM execution engine.

4.1 Informal introduction

We now provide a general and intuitive introduction to the GSM model.

As already pointed out, each *artifact type* is described by an *information model* and a *lifecycle model*. The former is a schema of all business-relevant information for that type. The latter, instead, specifies the milestones and stages of a GSM artifact schema, including their relationships (stage hierarchy, association of milestones to stages, and association of tasks to atomic stages), and the sentries that govern the guards and milestones.

An *artifact instance* is a business-relevant conceptual entity that progresses through some business operations. Practically, is simply a data instance conform to the information model of the artifact type it belongs to. The information model has two components, the *data attributes* and the *status attributes*. Data attributes represent data relevant to the business, while status attributes hold information about the progress of the artifact instance as it moves through the business operations.

A *stage* is a cluster of business-relevant activities that might be performed in connection with an artifact instance. Stages are organized into a hierarchy, as they can be *atomic* or *composite*. An atomic stage contains exactly one *task*, which corresponds to a unit of business-relevant work that is to be performed by an outside agent (either human or machine). A composite stage contains other (sub-)stages. Each stage "owns" one or more milestones, and the intuitive goal of executing a stage is to achieve one of these milestones. At a given moment in time a stage may have status *open*, which corresponds to when the stage is executing, or *closed*, which corresponds to when the stage is not executing. In the model, a stage may execute multiple times in sequence, but cannot have two occurrences that are executing simultaneously.

Tasks are invoked by artifact instances. When a task is invoked, the artifact instance provides input data from its information model, and when the task terminates the task output data is written into the artifact instance information model.

From a high level perspective, we conceptually model the world as two macro-

entities interacting each other: the *artifact system* and the *environment*. The *artifact system* contains all artifact instances and a so-called *artifact service center* (ASC) which manages the artifact system by creating instances and by mediating and managing the interaction among artifacts and between artifacts and the environment. The *environment* represents the external world, or, in other words, everything that is not modeled as an artifact.

Artifacts instances use *events* to exchange input and output data with the environment and among themselves. There are basically two kinds of events: *(i) business events* and *(ii) internal events*. While internal events correspond to when a milestone or a stage changes status, and hence are used to evolve artifacts' lifecycles, business events are used for the communication among artifacts and from artifacts and the environment and they are managed as a queue by the artifact service center.

In order to define milestones and guards, we need the notion of sentry. A *sentry* is an expression in a condition language, that can refer to incoming events, internal events, and the values of data and status attributes. Sentries provide the core of the ECA-like rules that govern the progress of an artifact instance. In practical settings, a sentry will have the form **on** ⟨*event*⟩ **if** ⟨*condition*⟩ **then** ⟨*action*⟩, where either the event or condition may be omitted.

Milestones are business-relevant operational objectives that can be achieved by an artifact instance, and they are associated to stages. In the artifact information model each milestones is a boolean attribute. Milestones have associated achieving sentries that determine situations under which the milestone becomes *achieved* and changes to status true, and invalidating sentries that determine situations under which the milestone becomes *invalidated* and change to status false.

Stages have *guards* that control when a stage should become open. Each guard is specified as a sentry.

Example 4.1.1. *Let us model the process of purchasing items via an e-commerce website in GSM. The main entity is the* shopping cart*, hence it is modeled as an artifact type. The data attributes of the information model contain the information of interest:* cartID*, the identifier of the shopping cart;* customerID *the identifier of the customer and* LineItems*, a collection of the items currently contained in the shopping cart. A graphical representation of the lifecycle is in Figure 4.1, where rounded boxes denote stages, non-rounded boxes are tasks, diamonds denote guards (the crossed guard is a special guard that evaluates to true upon an artifact instance creation) and circles represent milestones. On artifact instance creation (when, e.g., the customer performs the login) the* Filling *stage opens. This stage has a substage,* AddLineItem*, which opens when the customer asks for an item to be added to the cart. As this stage opens, the task* CreateLI *starts by creating the line item of the requested type and adding it to the cart. The customer can add more than one line item (and each time the* AddLineItem *opens) and when he is done he can submit the order, which triggers the* OrderSent *milestone and closes the* Filling *stage. When* OrderSent *is achieved, the* Processing *stage opens. It is composed by two sub-stages:* RequestPayment *that handles the payment, and* Shipping*, that contacts the delivery company. As the order is shipped, the* Shipped *milestone is achieved, the stage closes and the lifecycle of the shopping cart artifact ends.* ■

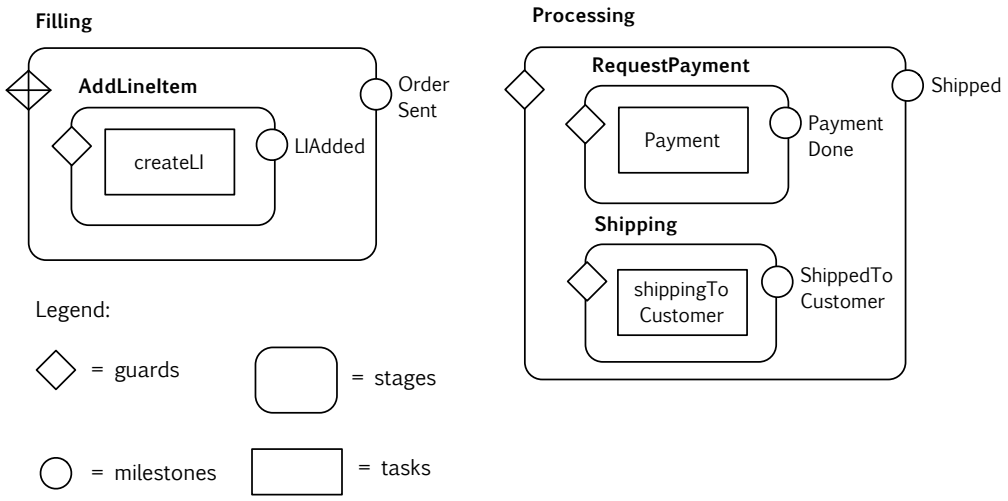


Figure 4.1. Graphical representation of the *Shopping Cart* artifact lifecycle described in Example 4.1.1.

4.2 Formal basis

This section formalizes the concepts introduced previously and presents the incremental semantics for GSM. We assume to have one artifact type only, the generalization to multiple artifact types is trivial.

Definition 4.1 (GSM data schema)**.** *A* GSM data schema *is a triple* $\mathcal{G}_d = (att, S, M)$ *where:*

- $att = att_s \cup att_d \cup LastIncEventType$ *is a finite set of attributes names, where* att_s *are the status attributes and* att_d *are the data attributes. Sets* att_s *and* att_d *are disjoint.*
- S *is a finite set of stage names;*
- $M \subset att_s$ *is a set of milestone names;*
- $\{active_s \mid s \in S\} \subseteq att_s$*, i.e., for each stage name* $s \in S$*, the status attribute* $active_s$ *occur in* att_s*.*

We also assume to have a set of event names EVENT and a domain Δ which include an undefined symbol $\perp$, a set of elements which represents numbers and strings, and the two boolean constants *true* and $false$.

Definition 4.2 (Snapshot)**.** *A snapshot of a GSM data schema is an assignment function from attribute names to the domain* $\Sigma : att \to \Delta \cup \text{EVENT}$ *where:*

- $\Sigma(a) \in \{true, false\}$ *for each* $a \in att_s$ *and*
- $\Sigma(LastIncEventType) \in \text{EVENT}$*.*

The alphabet of the condition language $\mathscr{L}$ is the set of attributes *att* of a GSM data schema, a finite set of event names EVENT and the boolean constant *true* and

$false$. The syntax of the language is formally presented in [96] and reporting it here is out of the scope of the thesis. Concerning the notation, when it is clear from the context that $A \in att$ is boolean, we use a and $\neg a$ to mean that the value of A is $a = true$ and $a = false$ respectively and we use $a_1, a_2, \ldots$ for the values of attributes names $A_1, A_2, \ldots$.

Variables of the language correspond to attributes in att and event names in EVENT. Hence, in order to evaluate a formula, we need to associate variables to $\Delta \cup$ EVENT. A snapshot can be though to as an assignment for the variables of a formula $\Phi \in \mathscr{L}$. Given a formula $\Phi \in \mathscr{L}$ we write $\Sigma \models \Phi(x_1 \ldots x_n)$ iff $\Phi(\Sigma(x_1) \ldots \Sigma(x_n))$ evaluates to true accordingly to the semantics of $\mathscr{L}$. Formulas in $\mathscr{L}$ can also be *temporal*, meaning that they can refer to different time instants, i.e., different snapshots of the system. Actually they can refer to only two snapshots, Σ and Σ', where we establish the convention, as customary in the verification community, that primed snapshots Σ' are constructed after Σ. Consequently, in formulas, we will use primed variables to be associated to elements in Δ according to Σ', and unprimed variables to be associated to elements in Δ according to Σ. Formally, given a formula $\Phi(x_1 \ldots x_n, x'_1 \ldots x'_m)$ where $x_1 \ldots x_n, x'_1 \ldots x'_m \in att$, the pair (Σ, Σ') satisfies Φ, denoted $(\Sigma, \Sigma') \models \Phi$ if $\Phi(x_1/\Sigma(x_1) \ldots x_n/\Sigma(x_n), x'_1/\Sigma'(x_1) \ldots x'_m/\Sigma'(x_m))$ evaluates to true, where $(x_1/\Sigma(x_1)$ substitues to x_1 the value $\Sigma(x_1)$ in Φ. Thinking in terms of temporal logics, e.g., LTL, we can consider a primed attribute a' as a $\mathbf{X}\ a$ where $\mathbf{X}$ is the LTL next temporal operator. We call *local* a formula Φ with no primed attributes.

Communications in GSM can take place in several ways. As a first classification, we can distinguish between *business events* and *status events*. We now analyze in detail the first class. *Business events* can be in turn separated into *one-way* and *two-way* events. *One-way* events are sent unsolicitedly from the environment to an artifact or from an artifact to another artifact.

An incoming *one-way (event) type* is a triple $E = (N, O, \psi)$, where $N \in$ EVENT is the event name, $O \subseteq att_d$ is the event payload structure and ψ is a condition (a local formula in $\mathscr{L}$) which refers to attributes in O.

A *one-way event instance*, or simply a one-way message event is a pair $e = (N, p)$ where $p : O \to \Delta$ is the payload such that $p \models \psi$. The condition ψ in a one-way event type formally represents restrictions on the output attributes.

Two-way events, instead, model a request and its related response. The request can come from the environment to an artifact (usually artifact creation) or from an artifact to the environment (task invocation) or finally, among artifacts. Notice that all these events are not synchronous but always stored in a queue, from where a random one is chosen at the beginning of each business step.

We assume a set TASK of *task names*, disjoint from the other sets of names already established. A *task* is a tuple (T, I, O, ψ) where $T \in$ TASK is a task name, $I \subseteq att_d$ are the input attributes, $O \subseteq att_d$ are the output attributes and ψ is a logical formula in $\mathscr{L}$ expressing the postconditions of the task. Given that the postcondition should refer to two different snapshots Σ and Σ', where Σ is the snapshot of the system when the task is invoked and Σ' is the next snapshot when it finishes, ψ refers to attributes in I without primes and attributes in O with

primes.

A *task invocation event type* is a pair $E = (T, I)$ where T is a task name and I are the input attributes of T. A *task invocation event instance* of type E is a tuple $e = (T, p)$ where $p : I \to \Delta$ is the input payload of the event.

A *task termination* event type is a triple $E = (T, I, O)$ where T is a task name, and I and O are the input and output attributes of T. A *task termination event instance* is a triple $e = (T, p, p')$ where (T, p) is a task invocation event instance, $p' : O \to \Delta$ is the output of the task and $(p, p') \models \psi$ evaluates to true. Here p is called the *input payload* of e and p' is called the *output payload* of e.

We now move to status events. A *status event* for a GSM data schema $\mathcal{G}_d$ is an expression of the form $\neg a \wedge a'$ or $a \wedge \neg a'$ where $a \in att_s$. To ease the notation, from now on we use $+a$ as a shortcut for $\neg a \wedge a'$ and $-a$ for $a \wedge \neg a'$. Thinking in terms of temporal logics, $+a \equiv \neg a \wedge \mathbf{X}\ a$ and $+a \equiv a \wedge \mathbf{X}\ \neg a$. Our condition language $\mathscr{L}$ can refer to status events as well, and the intuitive meaning is that $+a$ is true when a shifted from false to true during the course of a B-step, and analogously for $-a$.

A *sentry* for a GSM data schema $\mathcal{G}_d$ is a boolean formula of the form $\tau \wedge \gamma$, where:

- γ is a formula that contains no incoming event types or status events and
- τ is:
  - empty or
  - has the form $\mathsf{LastIncEventType} = E$ or
  - has the form $\{+, -\}a$ for some status attribute $a \in att_s$.

It is easy to transform a sentry $\tau \wedge \gamma$ into the classical form ***on*** $\langle event \rangle$ ***if*** $\langle condition \rangle$. The set of sentries for GSM_d is denoted SENTRY.

A GSM schema includes a GSM data schema and a GSM lifecycle.

Definition 4.3 (GSM model)**.** *A* GSM schema *is a tuple* $\mathcal{G} = (att, S, M, L)$ *where* $\mathcal{G}_d = (att, S, M)$ *is a GSM data schema and* $L = (Substages, Tasks, Owns, Guards, Ach, Inv)$ *is the lifecycle schema, where:*

- $Substages : S \to 2^S$ *is a function from* S *to finite subsets of* S*, such that the relation* $\{(s, s') \mid s' \in Substages(s)\}$ *creates a forest. A root of this forest is called a* top-level stage*, a leaf is called an* atomic stage*, and a non-leaf node is called a* composite stage*. The set of atomic stages is denoted* S_{atom}*.*
- $Task : S_{atom} \to$ TASK *is an injection.*
- $Owns : S \to 2^M \setminus \emptyset$ *is such that* $Owns(s) \cap Owns(s') = \emptyset$ *for* $s \neq s'$*. A stage* s *owns a milestone* m *if* $m \in Owns(s)$*.*
- $Guards : S \to 2^{\text{SENTRY}}$*.*
- $Ach : M \to 2^{\text{SENTRY}}$*. For* m *each element of* $Ach(m)$ *is called* achieving sentry *of* m*.*
- $Inv : M \to 2^{\text{SENTRY}}$*. For* m *each element of* $Inv(m)$ *is called* invalidating sentry *of* m*.*

A GSM data snapshot Σ is a snapshot for a GSM schema $\mathcal{G}$ if it satisfies the following *invariants*:

- GSM-1: Stage and milestone cannot both be true. If $m \in Owns(s)$ and $\Sigma(active_s) = true$, then $\Sigma(m) = false$.
- GSM-2: No activity in closed stage. If $\Sigma(active_s) = false$ for stage $s \in S$ and $s' \in Substages(s)$, then $\Sigma(active_{s'}) = false$.

We now formalize the notion of the environment. It is assumed that the environment performs external tasks, such as human tasks, that are invoked by the artifacts. To model the environment of a GSM schema it is sufficient to incorporate, for each task $T \in$ TASK variables that hold the values of the input variables used in an invocation of T. The *environment* for a GSM schema $\mathcal{G} = (att, S, M, L)$ is the set of attributes $att_{env} = \{A_T \in att \mid T = (T, I, O, \psi) \in Task(s), s \in S_{atom}, A_T \in I\}$. An *environment snapshot* is a function $\Sigma_{env} : att_{env} \to \Delta$.

Example 4.2.1. *We continue Example 4.1.1 by describing the schema more formally. The* Shopping Cart *artifact is a tuple* (att, S, M, L) *where* $L = (Substages, Tasks, Owns, Guards, Ach, Inv)$ *and:*

- $att = att_d \cup att_s \cup LastIncEventType$;
- $att_d =$ CartID, CustomerID, LineItems;
- $S =$ {Filling, AddLineItem, Processing, RequestPayment, Shipping};
- $M =$ {LIAdded, OrderSent, PaymentDone, ShippedToCustomer, Shipped};
- $att_s = M \cup \{active_{\text{Filling}}, active_{\text{AddLineItem}}, active_{\text{Processing}}, active_{\text{RequestPayment}}, active_{\text{Shipping}}\}$;
- $Substages($Filling$) = \{$AddLineItem$\}$, $Substages($Processing$) = \{$RequestPayment, Shipping$\}$;
- $Tasks($AddLineItem$) =$ createLI, $Tasks($Processing$) =$ Payment *and* $Tasks($Shipping$) =$ shippingToCustomer;
- $Owns($Filling$) =$ OrderSent, $Owns($AddLineItem$) =$ LIAdded, $Owns($Processing$) =$ Shipped, $Owns($RequestPayment$) =$ PaymentDone, $Owns($Shipping$) =$ ShippedToCustomer;
- $Guards($Filling$) =$ ***on*** *creation*, $Guards($AddLineItem$) =$ ***on*** *AddLI*, $Guards($Processing$) =$ ***on*** $+OrderSent$, $Guards($RequestPayment$) =$ ***on*** $+OrderSent$, $Guards($Shipping$) =$ ***on*** $+PaymentDone$;
- $Ach($OrderSent$) =$ ***on*** *SubmitOrder*, $Ach($LIAdded$) =$ ***on*** *createLITermination*, $Ach($Processing$) =$ ***on*** $+ShippedToCustomer$, $Ach($PaymentDone$) =$ ***on*** *PaymentTermination*, $Ach($ShippedToCustomer$) =$ ***on*** *shippingToCustomerTermination*

where we assume AddLI, SubmitOrder, createLITermination, PaymentTermination and shippingToCustomerTermination to be event names on EVENT.

4.3 GSM incremental semantics

The incremental semantics is presented in [83, 47] and it is centered around the concepts of GSM business steps (B-step) and Prerequisite-Antecedent-Consequent (PAC) rules. PAC rules are a variation of ECA rules used to capture the intended meaning of the sentries in a GSM schema, and also the GSM invariants. A B-step describes how a GSM snapshot Σ evolves to a new snapshot Σ' when an event instance e is consumed. Technically, starting from Σ, the status of the system (data attributes) evolves in a sequence of inconsistent snapshots Σ_i up to the new stable state Σ'. Each of the Σ_i is the result of applying a PAC rule from the previous snapshot Σ_{i-1}, and such a step is called *micro-step*. When no more PAC rules can be applied, that means that the last snapshot is returned as the result of the B-step. During each micro-step new events to send to the environment are possibly generated. Each PAC rule is associated to one or more GSM constructs (e.g. milestone) and has three components:

- **Prerequisite:** this part of a rule is considered relative to the initial snapshot Σ and determines whether the rule is *relevant* for an incoming event e and the given moment of time t.
- **Antecedent:** this part is considered relative to the current snapshot Σ_i and determines whether the rule is eligible for execution, or *executable*.
- **Consequent:** whenever the rule is executable, it may be nondeterministically be chosen to be fired in order to create Σ_{i+1} according to the consequent.

Intuitively, B-steps are considered to be unit of business relevant change, and they capture all the effects of incorporating a single incoming event, including changes to milestones and stage status.

A B-step is formally characterized by a tuple $(\Sigma, e, \Sigma', \Sigma_{env}, G, \Sigma'_{env})$ where the following hold:

- Σ is the previous GSM schema snapshot;
- e is an event instance of some event type $E \in \text{EVENT}$;
- Σ' is next GSM schema snapshot;
- Σ_{env} is the previous environment snapshot;
- G is the set of *generated event occurrences*, each one having a task invocation type, and they arise when atomic stages become active;
- Σ'_{env} is the next environment snapshot.

To illustrate the notion of B-step, we describe key aspects of the incremental formulation of the semantics. The next GSM schema snapshot Σ' is constructed in two phases. The first is to incorporate event e into Σ, by computing the "immediate effect" Σ^e of e on Σ. Intuitively, this has the effect of *(i)* updating the attribute $LastIncEventType$ to hold the type of e, and *(ii)* updating the values of all data attributes directly affected by the payload of e. In order to be consumed, an event e should be *applicable*. Let $\mathcal{G} = (att, S, M, L)$ be a GSM schema and Σ a snapshot of $\mathcal{G}$. Incoming event e is *applicable* to Σ if either

$e = (N, p)$ has type $E = (N, O, \psi) \in$ EVENT and $p \models \psi$ or $e = (T, p, p')$ has type $E = (T, I, O) \in$ EVENT, $T = (T, I, O, \psi) = Tasks(s)$ for some stage $s \in S$, $\Sigma(active_s) = true$ and $(p, p') \models \psi$.

The second phase is to incorporate the effects of the guards, achieving and invalidating sentries for milestones, and the two GSM invariants. A family of ECA-like rules corresponding to these constructs is derived from $\mathcal{G}$. A sequence $\Sigma = \Sigma_0, \Sigma^e = \Sigma_1, \Sigma_2 ... \Sigma_n = \Sigma'$ of so-called *pre-snapshots* is constructed, where *(i)* each step in the computation after Σ^e is called a micro-step and it corresponds to the application of one ECA-like rule, and where *(ii)* no ECA-like rule can be applied to Σ_n. Notice that the intermediate values $\Sigma_1, ..., \Sigma_{n-1}$ might violate GSM-1 or GSM-2 invariants, which is why they are permitted to be pre-snapshots as opposed to snapshots. There are restrictions on the ordering of rule application, as we will see later on. Finally, Σ' is returned as the result of the B-step. For each micro-step one also maintains a set G_i of generated events, which are sent to the environment at the termination of the B-step. When discussing B-steps we shall focus on triples of the form (Σ, e, Σ') where Σ, Σ' are snapshots of the GSM schema and e is an incoming event applicable to Σ. Although the construction of a snapshot Σ' from snapshot Σ and event e is treated as a black box from the perspective of the business, the values of Σ and Σ' are considered business relevant.

A prerequisite-antecedent-consequent (PAC) rule is a triple (π, α, γ) where π, the prerequisite, is a (local) formula in $\mathscr{L}$, i.e., it contains no primed variable, α, the antecedent, is a possibly temporal formula in $\mathscr{L}$ and γ, the consequent, is a *status change event* of the form $\{+, -\}a$ where $a \in att_s$. Table 4.1 shows templates for the six kinds of PAC rules associated with a GSM schema $\mathcal{G}$. Before giving the intuitive meaning of the such rules, we have to provide their semantics.

Let $P = (\pi, \alpha, \gamma)$ a PAC rule. First of all notice that, while π is a local formula, α and γ are temporal ones. Indeed, they contain status events $\{+, -\}a$, i.e., formulas $\neg a \wedge a'$ or $a \wedge \neg a'$ containing primed and unprimed attributes, and hence, in order to be evaluated, a pair of snapshots is required. The crucial point of the semantics of a B-step is understanding which, among the sequence $\Sigma_0, \Sigma_1 \ldots \Sigma_n$, such snapshots are. Recall that from the previous stable snapshot Σ, the first micro-step consists in incorporating the effect of the incoming event, obtaining the sequence $\Sigma_0 = \Sigma, \Sigma_1 = \Sigma^e$ and then each PAC rule results in a new snapshot Σ_{i+1}. Let assume we already fired some PAC rules and obtained the sequence $\Sigma_0, \Sigma_1, \ldots, \Sigma_i$, but Σ_i is not "stable" because we can still fire P. We say that P is applicable to the sequence $\Sigma_0, \Sigma_1, \ldots, \Sigma_i$ iff (1) $\Sigma_0 \models \pi$ and (2) $(\Sigma, \Sigma_i) \models \alpha$. The new snapshot Σ_{i+1} is build from Σ_i by changing the value of the attribute in γ, and hence Σ_{i+1} is such that $(\Sigma_i, \Sigma_{i+1}) \models \gamma$. The reason why primed attributes of α are evaluated in Σ_i is that Σ_i contains attributes that could have changed from Σ_0 to Σ_i by PAC rules that fired before P. This introduces the importance of an ordering between PAC rules, which will be explained later on and it also clarifies the difference between π and α: both of them represent preconditions for the application of the rule, but while π is tested over the previous stable snapshot Σ_0, disregarding all micro-steps that already happened, formula α takes into account the changes introduced by the application of the PAC rules that fired before P. Finally, γ represents the effect of the rule and indeed it is evaluated over last micro step Σ_i and new one Σ_{i+1}.

Name	Basis	Prerequisite	Antecedent	Consequent
PAC-1	Guard: for each stage s and for each guard ϕ of s. Include term $active_{s'}$ in the antecedent iff s' is parent of s.	$\neg active_s$	$\phi(\wedge active_{s'})$	$+active_s$
PAC-2	Milestone achiever: for each milestone m of stage s with achieving sentry ϕ.	$active_s$	ϕ	$+m$
PAC-3	Milestone invalidator: for each milestone m of stage s with invalidating sentry ϕ.	m	ϕ	$-m$
PAC-4	Guard invalidating milestone: for each guard ϕ of a stage s and for each milestone m not occurring in a top-level conjunct $\neg m$. Include term $active_{s'}$ in the antecedent iff s' is a parent of s.	m	$\phi(\wedge active_{s'})$	$-m$
PAC-5	For each milestone m of a stage s.	$active_s$	$+m$	$-active_s$
PAC-6	For each stage s child of s'.	$active_s$	$-active_{s'}$	$-active_{s'}$

Table 4.1. Templates for the six kinds of PAC rules.

We now show the two key intuitions underlying the notion of B-step. The first one is called *toggle once principle*. This states that in a B-step (Σ, e, Σ') each status value attribute can change at most once during that construction. If this were not the case, a given status attribute might change value inside the B-step, but those changes would not be visible from the starting and ending snapshots of the B-step. This is not desired since when looking at a sequence of B-steps from outside, the system's behaviors may be not intuitive to understand. In other words, that would mean that the smallest unit of computation, i.e., the B-step, is too coarse-grained for system analysis. Such a principle is enforced by the use of prerequisites in the PAC rules. In the formalism, moreover, enforcing the toggle once principle has the practical advantage of preventing infinite cycles in the incremental computation of a B-step.

The second intuitive principle is called *inertial*. This states that if a status attribute changes during a B-step, then there should be a "justification" that is visible by examining only the starting and ending snapshots of the B-step. Now, in the general case, the set of PAC rules of a GSM schema involves a form of negation. As is well-known from logic programming and datalog [99], the presence of negation in rules can lead to outcomes that are not inertial. In the GSM operational semantics this is avoided using an approach reminiscent of stratification as developed in those fields [4, 71]. In particular, the approach involves requiring that a certain relation defined on the rules is acyclic, and requiring that the order of rule firing complies with that relation. In order to enforce the second principle, we introduce the notion of *polarized dependency graph*.

The purpose of the *polarized dependency graph* is twofold: *(i)* to capture dependencies between PAC rules in order to contraints their application during the incremental construction of B-step and *(ii)* to check whether a GSM schema is well-formed. Roughly speaking, nodes of a polarized dependency graph are polarized attributes $\{+,-\}a$ and an edge from $\{+,-\}a_1$ to $\{+,-\}a_2$ is included in the graph if consideration of PAC rule triggering a_2 should be performed only after consideration of PAC rule triggering a_1.

Definition 4.4. *Polarized dependency graph. A* polarized dependency graph *of a GSM schema $\mathcal{G}$, denoted by $PDG(\mathcal{G})$ is defined as follows. For each status attribute $a \in att$, we have two nodes $+a$ and $-a$. For each stage s and for each guard ϕ we have a node $+s.\phi$. In the following, the antecedent α of a PAC rule is written $\tau \wedge \gamma$ where to τ and γ applies the same restrictions of the sentries.*

- *For each PAC-1 rule $(\neg active_s, \tau \wedge \gamma, +active_s)$:*
  - *if $\{+,-\}a$ is a polarized attribute occurring in τ, include directed edge $(\{+,-\}a, +s.\phi)$.*
  - *if a status attribute b occurs in γ, then include two directed edges $(+b, +s.\phi)$ and $(-b, +s.\phi)$.*
- *For each guard ϕ of stage s:*
  - *add edge $(+s.\phi, +active_s)$;*
  - *for each milestone m owned by s that does not occur in a top-level conjunct of form $\neg m$ in γ, add the edge $(+s.\phi, -m)$ (this edge corresponds to PAC-4).*

- *For each PAC rule* $(\pi, \tau \wedge \gamma, \{+,-\}a)$ *from templates PAC-2 or PAC-3:*
  - *if* $+b$ *is a polarized attribute occurring in* τ*, then include directed edge* $(\{+,-\}b, \{+,-\}a)$*;*
  - *if* b *is a status attribute occurring in* γ*, then add two edges* $(+b, \{+,-\}a)$ *and* $(-b, \{+,-\}a)$*.*
- *for each PAC-5 rule* $(active_s, +m, -active_s)$ *add edge* $(+m, -active_s)$ *and*
- *for each PAC-6 rule* $(active_s, -active_{s'}, -active_s)$*, ad edge* $(-active_{s'}, -active_s)$*.*

Definition 4.5. *The* status only *polarized dependency graph, denoted* $PDG^s(\mathcal{G})$ *is formed from* $PDG(\mathcal{G})$ *as follows:*

- *the set of nodes of* $PDG^s(\mathcal{G})$ *is the same as* $PDG(\mathcal{G})$*;*
- *each edge of* $PDG(\mathcal{G})$ *that does not involve a guard is included and*
- *for each pair of edges* $(\{+,-\}a, +s.\phi)$ *and* $(+s.\phi, \{+,-\}b)$ *add edge* $(\{+,-\}a, \{+,-\}b)$*.*

Definition 4.6. *A GSM schema* $\mathcal{G}$ *is* well-formed *if* $PDG^s(\mathcal{G})$ *is acyclic.*

From a polarized dependency graph we can extract the order of application of PAC rule, in order to ensure the principles which the semantic of B-step is based on and, in particular, the inertial principle. By construction, there is an edge between two nodes $(\{+,-\}a, \{+,-\}b)$ in a PDG if the change in the value of a triggers a change in b. That means that the PAC rule that changes the value of a should be fired *before* the PAC rule that changes the value of b. Following such dependencies, given a polarized dependency graph $PDG(\mathcal{G})$ for a GSM schema $\mathcal{G}$, and assumed that $\mathcal{G}$ is well-formed, we obtain a partial ordering between the PAC rules. It is sufficient to choose a total order that satisfies such a partial order to guarantee that the principles are satisfied.

4.4 Reduction to relational artifact-centric processes

This section shows that relational data-centric dynamic systems are powerful enough to simulate the execution of GSM schemas. To do so, we provide a translation procedure which rewrites a GSM schema into a DCDS schema.

To ease the presentation, we assume an artifact type only. The extension to multiple types is trivial.

Let $\mathcal{G} = (att, S, M, L)$ be a GSM schema with $\mathcal{G}_d = (att, S, M)$ and $L = (Substages, Tasks, Owns, Guards, Ach, Inv)$, we show how to encode $\mathcal{G}$ as a DCDS $\mathcal{S} = (\mathcal{D}, \mathcal{P})$ where $\mathcal{D} = (\mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0)$ and $\mathcal{P} = (\mathcal{F}, \mathcal{A}, \varrho)$ are as in Section 3.5. We assume to have a set EVENT of event names and a set of PAC-rules PAC.

The set $\mathcal{R}$ is composed by:

- $R_{att} = (id, a_1 \dots a_n, s_1 \dots, s_m, m_1 \dots m_\ell)$ is a relation which represents the data and status attributes of artifacts in $\mathcal{G}_d$. With a slight abuse of notation we use m to denote the number of states and m_i to denote a milestone. Attribute id is used to distinguish between artifact instances; $a_1 \dots a_n$ are the

data attributes in att_d and the status attributes in att_s are $s_1 \dots s_m$, with $m = |S|$ storing the status of each stage (open or close) and $m_1 \dots m_\ell$ with $\ell = |M|$ store the status of each milestone.

- $R_{m_1} \dots R_{m_\ell}$ is a set of relations, one for each milestone in M. Each relation $R_{m_i} = (id, newState)$ stores the fact that a milestone m_i of artifact instance id has been recently achieved or invalidated. Such relations are used to keep track of the internal events concerning milestones occurring during a B-step.
- $R_{s_1} \dots R_{s_m}$ is a set of relations, one for each stage in S. Each relation $R_{s_i} = (id, newState)$ has the same purpose of the previous R_{m_i}.
- $R^{inc}_{E_1} \dots R^{inc}_{E_k}$ is a set of relations, one for each event type $E \in \text{EVENT}$, where $k = |\text{EVENT}|$. Each relation $R^{inc}_{E_i} = (id, p_1 \dots p_t)$ stores the id of artifact instance that is the receiver of the event and $p_1 \dots p_t$ is the payload of the event and can be different for each event type. Such a relation is used to store the type and the payload of the event that is currently being consumed.
- $R^{out}_{E_1} \dots R^{out}_{E_k}$ is a set of relation, one for each event type $E \in \text{EVENT}$, where $k = |\text{EVENT}|$. Each relation $R^{out}_{E_i} = (id, p_1 \dots p_t)$ stores the id of the artifact that generated the event and $p_1 \dots p_t$ is the payload of the event and can be different for each event type. Such a relation is used to store the events generated by artifact instances during a B-step that have to be sent to the environment.
- $R_{sync}(bool_1, bool_2)$ is an auxiliary relation needed to synchronize actions within a B-step.
- $R_{PAC} = (id, x_1 \dots x_d)$ is a relation where each x_i is a boolean flag that is true if artifact instance id has already considered the i-th PAC rule in PAC during a B-step. This relation is used to implement the ordering among PAC rules during a B-step.

We now move to the dynamic aspects of DCDS $\mathcal{S}$, namely the actions and the process. Our aim is to mimic the incremental semantics of GSM, i.e., to have actions and a process that simulate each micro-step of a B-step in a GSM schema. In order to do so, the set of actions in $\mathcal{A}$ is composed by:

- one action for each PAC rule in PAC;
- one action for each incoming event, to simulate the immediate effect of incorporating an event;
- one action to send outgoing events at the end of a B-step;
- one action to create an artifact instance;
- one action to remove an artifact instance.

Recalling the semantics of B-steps, the first micro-step of a B-step is the incorporation of an incoming event into the current snapshot, then a sequence of micro-steps each corresponding to the firing of a PAC rule and finally a micro-step which send all the generated events to the environment. We will encode each micro-step as a separate condition-action rule in the process ϱ of the DCDS schema such that the progression of such a system simulates the evolution of the GSM schema. In order

to do so, we should observe the semantic assumption of a GSM model, namely: *(i)* "one-message-at-a-time" and "toggle-once" principles, *(ii)* the finiteness of micro-steps within a B-step, and *(iii)* their order imposed by the polarized dependency graph. Such constraints are met by making use of the auxiliary relations introduced before as follows:

- we restrict the relational artifact system to process only one incoming message at a time by make use of the auxiliary relation $R_{sync}(bool_1, bool_2)$: $R_{sync}(true, true)$ means that the system is in a stable snapshot and hence a new event can be consumed; $R_{sync}(false, true)$ means that an incoming event has been chosen and next step to be performed is the event incorporation and $R_{sync}(false, false)$ means that the event has been incorporated and actions corresponding to PAC rules are ready to fire.
- In order to ensure "toggle once" principle and guarantee the finiteness of sequence of micro-steps triggered by an incoming event, we use the relation $R_{PAC}(id, x_1, ..., x_c)$, where each boolean x_i encodes whether the corresponding PAC rule is eligible to fire for artifact instance id at a given moment in time, namely, in a specific micro-step. The initial setup of the eligibility tracking flags is performed at the beginning of a B-step, based on the evaluation of the prerequisite of each PAC rule. When $x_i = false$, the corresponding CA-rule is eligible to apply and has not yet been considered for application. When instead $x_i = true$, then either the rule has been fired, or its prerequisite turned out to be false. R_{PAC} is also used to enforce the firing order of CA-rules (corresponding to PAC rule) induced by the dependency graph of $\mathcal{G}$ as follows: for each CA-rule $Q \mapsto \alpha$ corresponding to a given PAC rule P, condition Q is in conjunction with a further formula, used to check whether all the PAC rules that precede ρ according to the ordering imposed by the dependency graph have been already processed. More specifically, the CA-rule becomes $Q \wedge exec(P) \mapsto \rho$, where $exec(P) = \bigwedge_i x_i$ for all indexes i of PAC rules that precede P.

We now show the set of actions needed for simulating a GSM schema. The main intuition underlying the translation is that the output attributes of tasks in the GSM schema are modeled as task invocations in the relational data-centric dynamic systems.

In the GSM model, the selection of the event is not captured, and a B-step starts with the incorporation of the chosen event. In a DCDS, instead, we have to explicitly model the choice of an event. We hence need a special action for each incoming event type E that models the request, to the external world, for an event of type E to be consumed. It is worth stressing that this action does not correspond to any micro-steps in the GSM model.

$$
\begin{aligned}
R_{sync}(true, true) \mapsto\ & \mathsf{askForEventE_i}() : \{ \\
& (1)\ \ true \rightsquigarrow R^{inc}_{E_i}(f^{E_i}_1(\ldots)\ldots f^{E_i}_c(\ldots)) \\
& (2)\ \ true \rightsquigarrow R_{sync}(false, true) \\
& (3)\ \ [\mathsf{CopyAllOtherRelations}]\ \}
\end{aligned}
$$

The precondition $R_{init}(true)$ is used to be sure we are at the beginning of a B-step, i.e., that we are allowed to select an incoming message to be incorporated. Effect (1) is used to store the chosen event of type E_i in the special relation R_{E_i} that will be read by the other actions during the B-step, while effect (2) blocks other actions of type askForEvent to be fired until the current B-step ends. As we said before, we assume the first attribute of relation R_{E_i} to be the id of the artifact instance the event is addressed to.

We now consider the action corresponding to the first micro-step of a B-step, i.e., the incorporation of the chosen event in the system. For each event type E_i we have the following CA rule:

$$\exists \boldsymbol{a}, \overline{\boldsymbol{a}}, \boldsymbol{s}, \boldsymbol{m}. R_{att}(id, \boldsymbol{a}, \boldsymbol{s}, \boldsymbol{m}) \wedge R^{inc}_{E_i}(id, \overline{\boldsymbol{a}}) \wedge R_{sync}(false, true) \mapsto \mathsf{ImmEffE_i}(id) : \{$$

(1) $R_{att}(id, \boldsymbol{a}, \boldsymbol{s}, \boldsymbol{m}) \wedge R^{inc}_{E_i}(id, \overline{\boldsymbol{a}}) \rightsquigarrow R_{att}(id, \overline{\boldsymbol{a}}, \boldsymbol{s}, \boldsymbol{m})$

(2) $true \rightsquigarrow R_{sync}(false, false)$

(3) for each PAC rule :

$\pi_j(id) \rightsquigarrow R_{PAC}(id, x_1 \dots x_j = false \dots x_d)$

$\neg\pi_j(id) \rightsquigarrow R_{PAC}(id, x_1 \dots x_j = true \dots x_d)$

(4) [CopyAllOtherRelations] }

where $\boldsymbol{a} = a_1 \dots a_n$, and analogously for $\overline{\boldsymbol{a}}$, $\boldsymbol{s}$ and $\boldsymbol{m}$ and where $\pi_i(id)$ is the precondition of the i-th PAC rule of the GSM schema. Effect (1) incorporates the payload $\overline{\boldsymbol{a}}$ into the data attributes of artifact instance id. The status attributes $\boldsymbol{s}$ and $\boldsymbol{m}$ do not change yet, but they possibly will as a result of other actions, namely those corresponding to PAC rules. Effect (2) blocks other $\mathsf{ImmEffE_i}$ actions to fire, while effects (3) select the PAC rules that will fire within the current B-step by checking their preconditions. The only difference between action $\mathsf{ImmEffE_i}$ for event type E_i and another action $\mathsf{ImmEffE_k}$ for event type E_k is in the number and type of Skolem functions in the effects.

Once an event had been incorporated, next micro-steps are applications of PAC rules. In what follows we provide, for each PAC rule template in the GSM model, the corresponding CA rule to be used in a relational artifact-based system.

For each stage $s_j \in S$ and for each guard $g = \tau \wedge \gamma$ there is a PAC-1 rule (see Table 4.1) that corresponds to the following CA rule:

$$\exists \boldsymbol{x}. R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \wedge \mathsf{exec}(i) \mapsto \mathsf{PAC_i}(id) : \{$$

(1) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge \mathsf{R}_\tau(id, \boldsymbol{a}_2) \wedge \gamma \rightsquigarrow R_{att}(id, \boldsymbol{a}_1, s_1 \dots s_j = true \dots s_m, \boldsymbol{m})$

(2) $\mathsf{R}_\tau(id, \boldsymbol{a}_2) \wedge \gamma \rightsquigarrow R_{s_j}(id, true)$

(3) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge \mathsf{R}_\tau(id, \boldsymbol{a}_2) \wedge \gamma \rightsquigarrow R^{out}_{E_k}(id, \boldsymbol{a}_1)$

(4) $R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \rightsquigarrow R_{PAC}(id, x_1 \dots x_i = true \dots x_d)$

(5) [CopyAllOtherRelations] }

where:

- $\mathsf{exec}(i)$ is a shortcut for $\bigwedge_j x_j$ for all $1 < j < i$, which checks whether all PAC rules that have to fire before the i-th (according to the polarized dependency graph) have already been executed and

- $\mathsf{R}_\tau(id, \boldsymbol{a}_2) = R^{inc}_{E_k}(id, \boldsymbol{a}_2)$ if τ contains an incoming event condition or $\mathsf{R}_\tau(id, \boldsymbol{a}_2) = R_{m_1}(id, newState) \wedge \ldots \wedge R_{s_1}(id, newState) \ldots$ if τ contains internal event conditions.

Effect (1) changes the status attribute of the stage that is opening; effect (2) propagates such an event as an internal message; effect (3) stores event to be sent at the environment at the end of the B-step and effect (4) blocks the same action to be fired again during the same B-step. For the sake of readability we did not include the condition for checking whether the parent stage s' is open, but it is enough to add a conjunct to effects (1), (2) and (3) to achieve the purpose.

For each stage $s \in S$ and milestone m of s with achieving sentry $\tau \wedge \gamma$, there is a PAC-2 rule that corresponds to the following CA rule:

$\exists \boldsymbol{x}.R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \wedge \mathsf{exec}(i) \mapsto \mathsf{PAC_i}(id) : \{$

(1) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge \mathsf{R}_\tau(id, \boldsymbol{a}_2) \wedge \gamma \rightsquigarrow R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, m_1 \ldots m_j = true \ldots m_\ell)$

(2) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge \mathsf{R}_\tau(id, \boldsymbol{a}_2) \wedge \gamma \rightsquigarrow R^{out}_{E_k}(id, \boldsymbol{a}_1)$

(3) $\mathsf{R}_\tau(id, \boldsymbol{a}_2) \wedge \gamma \rightsquigarrow R_{s_j}(id, true)$

(4) $R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \rightsquigarrow R_{PAC}(id, x_1 \ldots x_i = true, x_d)$

(5) [CopyAllOtherRelations] $\}$

where $\mathsf{exec}(i)$ and $\mathsf{R}_\tau(id, \boldsymbol{a}_2)$ are as before.

Concerning the family of PAC-3 rules, one for each milestone invalidating condition, the corresponding CA rule template is as before but for the effect (2) where $m_j = false$ instead of $m_j = true$.

The template of PAC-4 rule is used for invalidating milestones of a stage when it opens. For each stage s_k, guard $g = \tau \wedge \gamma$ of s_k and milestone m_j of s_k we have the CA rule:

$\exists \boldsymbol{x}.R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \wedge \mathsf{exec}(i) \mapsto \mathsf{PAC_i}(id) : \{$

(1) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge R_{s_k}(id, true) \rightsquigarrow R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, m_1 \ldots m_j = false \ldots m_\ell)$

(2) $R_{s_k}(id, true) \rightsquigarrow R_{m_j}(id, false)$

(3) $R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \rightsquigarrow R_{PAC}(id, x_1 \ldots x_i = true, x_d)$

(4) [CopyAllOtherRelations] $\}$

where $\mathsf{exec}(i)$ is as before and where effect (1) invalidates the milestones of S_k acting on the status attributes; effect (2) propagates the invalidation of the milestone as an internal event; and effect (3) is as before.

PAC-5 rules close a stage s when one of its milestone is achieved. For each stage s_k and each milestone m_j, we have the CA rule:

$\exists \boldsymbol{x}.R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \wedge \mathsf{exec}(i) \mapsto \mathsf{PAC_i}(id) : \{$

(1) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge R_{m_j}(id, true) \rightsquigarrow R_{att}(id, \boldsymbol{a}_1, s_1 \ldots s_k = false \ldots s_m, \boldsymbol{m})$

(2) $R_{m_j}(id, true) \rightsquigarrow R_{s_k}(id, false)$

(3) $R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \rightsquigarrow R_{PAC}(id, x_1 \ldots x_i = true, x_d)$

(4) [CopyAllOtherRelations] $\}$

where effect (1) modifies the status attribute corresponding to stage s_k; effect (2) propagates the closure of s_k as an internal event and effect (3) is as before.

PAC-6 rules are used to close all child stages of s if it closes. For each stage s_j child of s_k we have the PAC rule:

$\exists \boldsymbol{x}.R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \wedge \mathsf{exec}(i) \mapsto \mathsf{PAC_i}(id) : \{$
(1) $R_{att}(id, \boldsymbol{a}_1, \boldsymbol{s}, \boldsymbol{m}) \wedge R_{s_k}(id, false) \rightsquigarrow R_{att}(id, \boldsymbol{a}_1, s_1 \dots s_j = false \dots s_m, \boldsymbol{m})$
(2) $R_{s_k}(id, false) \rightsquigarrow R_{s_j}(id, false)$
(3) $R_{PAC}(id, \boldsymbol{x}) \wedge \neg x_i \rightsquigarrow R_{PAC}(id, x_1 \dots x_i = true, x_d)$
(4) [CopyAllOtherRelations] $\}$

More actions are needed in the relational artifact-centric system to simulate the GSM model. In particular, after the firing of actions that simulate the GSM PAC rules, we still need to send the messages to the environment and unblock the askForEvent actions for the next B-step. This is done by the following CA rule:

$\exists \boldsymbol{x}.R_{PAC}(id, \boldsymbol{x}) \bigwedge_i x_i \mapsto \mathsf{final}(id) : \{$
(1) $true \rightsquigarrow R_{sync}(true, true)$
(2) [CopyAllOtherRelations] *but* $R_E^{out}, R_E^{inc} R_S, R_M, R_{PAC}$ $\}$

Sending the events to the environment is simulated by flushing the $R_{E_i}^{out}$ relation. All other relations needed during the B-step, i.e., relations for internal events R_m and R_s, the relation R_E^{inc} for the incoming event (consumed by now) and the relation R_{PAC} used for synchronize the actions, are no longer needed and not copied as well.

Lastly, special actions are required for managing an incoming event which asks for creating or destroying an artifact instance.

$R_{sync}(true, true) \mapsto \mathsf{askForCreation}() : \{$
(1) $true \rightsquigarrow R_{cr}(f_1^{cr}(\dots) \dots f_n^{cr}(\dots))$
(2) $true \rightsquigarrow R_{sync}(false, true)$
(3) [CopyAllOtherRelations] $\}$

analogously to $\mathsf{askForEventE_i}$ chooses a create artifact event to be consumed, and

$\exists \boldsymbol{a}.R_{cr}(id, \boldsymbol{a}) \wedge R_{sync}(false, true) \mapsto \mathsf{ImmEffCreation}(id) : \{$
(1) $R_{cr}(id, \boldsymbol{a}) \rightsquigarrow R_{att}(id, \boldsymbol{a}, false \dots false)$
(2) $true \rightsquigarrow R_{sync}(false, false)$
(3) for each PAC rule :
$\pi_j(id) \rightsquigarrow R_{PAC}(id, x_1 \dots x_j = false \dots x_d)$
$\neg\pi_j(id) \rightsquigarrow R_{PAC}(id, x_1 \dots x_j = true \dots x_d)$
(3) [CopyAllOtherRelations] *but* R_{cr} $\}$

which creates the new artifact instance (effect (1)). Notice that the id of the new instance is $f_1^{cr}(\dots)$.

Concerning the deletion we have the action:

$$R_{bock}(true, true) \mapsto \mathsf{askForDeletion}() : \{$$
$$(1)\ \ true \rightsquigarrow R_{del}(f_1^{del}(\ldots))$$
$$(2)\ \ true \rightsquigarrow R_{sync}(false, true)$$
$$(3)\ \ [\mathsf{CopyAllOtherRelations}]\ \}$$

which selects a delete artifact event and

$$R_{del}(id) \wedge R_{sync}(false, true) \mapsto \mathsf{ImmEffDeletion}(id) : \{$$
$$(1)\ \ \exists \boldsymbol{a}, \boldsymbol{s}, \boldsymbol{m}.R_{att}(id, \boldsymbol{a}, \boldsymbol{s}, \boldsymbol{m}) \wedge \neg R_{del}(id) \rightsquigarrow R_{att}(id, \boldsymbol{a}, \boldsymbol{s}, \boldsymbol{m})$$
$$(2)\ \ true \rightsquigarrow R_{sync}(true, true)$$
$$(3)\ \ [\mathsf{CopyAllOtherRelations}]\ but\ R_{del}, R_{att}\ \}$$

which copies all artifacts whose id does not match with the one to be deleted (effect (1)).

We now discuss the intuitions behind the translation. The interesting part is how we model the choice of incoming events. Actually, we *ask* the environment for events by means of action askForEvent. This is somehow not intuitive, given that in practical applications, like in the Barcelona prototype, a queue of incoming event is stored in the system, and at the beginning of each step a message is chosen according to any policy. Such a choice, however, allows us to simplify the mechanism of incorporating incoming events from the environment. An alternative solution would be to consider an action (with no preconditions) that checks at each micro-step the presence of new events from the environment and stores them in a specific relation, and another action choosing one of them at the beginning of the B-step. We believe this solution to be more cumbersome. In both cases, it is the system that asks for events, given that the only way for modifying relations are through actions. Besides, askForEvent is the only action that includes Skolem functions. Indeed, in line with the assumptions and the framework presented in Chapter 3, Skolem functions model the new values in the payload of incoming events. While in the GSM model the distinction between new and old values is blurred, in our framework it is crucial in order to prove decidability of verification. The remaining actions just implements the mechanism to order PAC rules and re-arrange (old) values in the information model.

In order to simulate the execution of a GSM schema, we need evaluations that associate functions to values of the domain.

Let $\mathcal{G}$ be a GSM schema with initial snapshot Σ_0 and $\mathcal{S}$ the DCDS obtained with the above translation, with initial state s_0. We prove that the transition system $\Upsilon_{\mathcal{G}}$ representing all possible executions of $\mathcal{G}$ starting from Σ_0 is equivalent to the execution transition system $\Upsilon_{\mathcal{S}}$ of $\mathcal{S}$ rooted in s_0. Let us analyze such transition systems. $\Upsilon_{\mathcal{G}}$ is obtained by iteratively applying the incremental semantics starting from Σ_0 and nondeterministically considering each possible incoming event. States of $\Upsilon_{\mathcal{G}}$ correspond to stable snapshots of $\mathcal{G}$, and each transition corresponds to a B-step (recall that the incremental semantics abstracts away the micro-steps as not relevant from the business viewpoint). Similarly $\Upsilon_{\mathcal{S}}$ is obtained starting from

s_0, and iteratively applying nondeterministically the CA-rules of the process, and their corresponding actions, in all possible ways. States of $\Upsilon_{\mathcal{S}}$ corresponds to pre-snapshots, as each action represents a micro-step. We prune such intermediate states and we connect two states provided that there is a sequence of micro-steps linking the two. Notice that stable states can be easily recognized as they are the only including tuple $(true, true)$ into relation R_{sync}. It can be shown (see [124] for details) that $\Upsilon_{\mathcal{G}}$ and $\Upsilon_{\mathcal{S}}$ after the pruning are equal. The proof develops into three main lemmas: the first one demonstrates that for each micro-step of $\mathcal{G}$ there is a condition-action rule in $\mathcal{S}$ that results in the same pre-snapshot; the second one that for each B-step in $\mathcal{G}$ there is a corresponding sequence of actions that leads to the same new snapshot and in the last one for each execution path in G is proven to correspond to a path in $\mathcal{S}$.

4.5 Discussion

In this Chapter we introduced the GSM model for data-aware business processes, and we showed a translation from GSM schemas to DCDSs. We proved that DCDS is a powerful formalism that is able to simulate artifact-centric models, such as GSM. Actually, it can be proven (see [124]) that Turing machines can be captured in GSM, and hence also in DCDS. By reduction to the halting problem, it can be easily shown that even verification of very simple properties, e.g., reachability, is undecidable.

In [11] both syntactic and semantic properties to achieve decidability of verification are shown. To verify a GSM model it is then enough to translate it into a DCDS, check for the syntactic properties to be verified and, if met, performing the verification. However in [124], along with the reduction of GSM to DCDS, a syntactic property that guarantees decidability of verification is provided. This is extremely useful because the reduction to a DCDS for verification purposes is no longer needed. The syntactic requirement is obtained by bringing the semantic property of state boundedness introduced in Section 3.5 to GSM: it amounts to have no "create-artifact" tasks in the GSM schema, as they can be the only source of state unboundedness. Indeed, one-way incoming messages, as well as other service call returns, do not increase the size of the data stored in the GSM information model. During micro-steps the information model does not grow again, as status attributes are just modified and lastly, the number outgoing event produced during a B-step is bounded and flushed at the end of the step anyway.

Given that GSM lacks of support for verification, others investigated techniques to fill this gap. In [19] the framework presented in [18] is extended to model GSM processes as multi-agent systems in order to use the same abstraction techniques to verify first-order temporal properties with epistemic modalities.

A first effort to develop a tool for verifying GSM schemas is GSMC [76]. The GSMC parses the XML specification of a GSM schema as produced by the Barcelona web editor and translates it into a symbolic transition systems that is used for symbolic model checking. It is not yet sound nor complete given that, for achieving decidability, data are finitely bounded and only one instance per artifact type is considered.

Chapter 5

Colored Petri nets as a concrete data-aware process model

This chapter introduces Colored Petri nets (CPNs), a well-known model for formalizing business processes and shows the main intuitions behind an informal translation of CPNs into DCDSs. As in the previous Chapter, we aim at validating the hypothesis that the foundational aspects of the relational artifact-centric systems presented in Chapter 3 have enough expressing power to model real world scenarios and that DCDSs can, in principle, be used as a underlying formalism for higher level models, as PNs and CPNs do (cf. Section 1.1.1).

The choice of colored Petri nets in this Chapter rather than other formalisms for modeling business processes are several. First of all, PNs have a precise mathematical foundation. This is a common point with our artifact-centric systems and indeed, it allows for formal investigations. For this reason, PNs have been chosen to develop semantics (more or less implicitly) for business processes modeling languages, such as EPCs, UML activity diagrams, WS-BPEL, workflow nets and YAWL. Another advantage is that the effects of activities (or transitions, or tasks or actions) are explicitly modeled in the formalism, i.e., *what* is done by activities is known in the model, as it is in our artifact-centric systems. Such a feature is not trivial: there can be advantages in not formalizing what actions do. Indeed, some modeling languages are interested in describing, among other things, human activities, whose outcome is unknown. This is, e.g., the case of the GSM model introduced in Chapter 4. On the other side, formal analysis, such as service composition or model checking of such systems is very challenging. Usually, a deep formalization is needed when offline analysis is more important than the actual execution of the model. For problems in artificial intelligence, e.g., planning, usually a strong formalization is needed, because the model of the system is not intended to be executed, but used to find a solution. Formalizing a business process is, instead, slightly different, because a tradeoff between offline analysis and executability is of interest. Seen in this perspective, our work is valuable also in artificial intelligence field (see Section 1.3). Third reason PNs are here analyzed is because they are a purely process-centric formalism where the need of data has been perceived as a strong limitation. Colored PNs have been actually proposed to overcome such a limitation. However, even though they are more expressive than classical PNs

[129] (actually they are Turing-complete) they cannot be considered a data-centric model, given that data assume no primary role in the modeling phase and they have no lifecycle. Nonetheless, there are similarities between CPN and artifact-centric systems, and the scope of this chapter is to compare the two.

Lastly, given the solid mathematical foundations of CPN, several attempts to perform formal analysis on them have been carried out, which is another point in common with our approach.

The technical definition and concepts of CPN are taken from [84].

5.1 Introduction to colored Petri nets

In standard Petri nets, tokens can model different entities, such as physical objects, information objects, collections of objects, states and conditions, but it is not possible to describe attributes of a token. In colored Petri nets, there are different types of tokens depending on the information they carry, and each token has a value conforming to its type. Describing the status, i.e., the marking, of a CPN is complex because, for each place and for each specific value of the place type, the number of token with that value should be provided. Indeed, the formal theory of colored Petri Nets bases on the concept of multiset, which is used in almost all formal definitions of a CPN.

Definition 5.1 (Multiset)**.** *A* multiset *over a non-empty set S is a total function $m : S \to \mathbb{N}$ which maps each element s to the number $m(s)$ of appearances of s in S.*

In order to enable a transition t in a CPN, not only each input place of t must contain enough tokens, but also additional contraints on the values of tokens can be included. Given that the contents of places are described by multisets, to formalize the notion of enabling and occurrence of transition and steps, we define operations on multisets.

Definition 5.2 (Multisets operations)**.** Membership*,* addition*,* scalar multiplication*,* comparison *and* size *are defined as follows, where m, m_1 and m_2 are multisets and $n \in \mathbb{N}$:*

- $\forall s \in S : s \in m := m(s) > 0$*;*
- $\forall s \in S : (m_1 +\!\!+\ m_2)(s) = m_1(s) + m_2(s)$*;*
- $\forall s \in S : (m_1 **\ m_2)(s) = m_1(s) * m_2(s)$*;*
- $(m_1 \ll= m_2)(s) \leftrightarrow \forall s \in S m_1(s) \leq m_2(s)$*;*
- $|m| := \sum_{s \in S} m(s)$.

A multiset m is infinite *if $|m| = \infty$, otherwise is* finite*. When $m_1 \ll= m_2$,* subtraction *is defined as:*

- $\forall s \in S : (m_1 -\!\!-\ m_2)(s) = m_1(s) - m_2(s)$.

We can now define the syntax of a colored Petri net.

Definition 5.3 (Colored Petri net)**.** *A* non-hierarchical Petri net *is a nine-tuple* $CPN = (P, T, A, 2^{\Sigma}, V, C, G, E, I)$ *where:*

- P *is a finite set of* places*;*
- T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*;*
- $A \subseteq P \times T \cup T \times P$ *is a set of* directed arcs*;*
- 2^{Σ} *is a finite set of non-empty* color sets*, where* Σ *is a set of colors;*
- V *is a finite set of* typed variables *such that* $Type(v) \in 2^{\Sigma}$ *for all variables* $v \in V$*;*
- $C : P \to 2^{\Sigma}$ *is a* color set function *which assigns a color set to each place;*
- $G : T \to \mathrm{EXPR}_V$ *is an* guard function *which assigns a guard to each transition* t *that evaluates to true or false, i.e.,* $Type(G(t)) = Bool$*;*
- $E : A \to \mathrm{EXPR}_V$ *is an* arc expression function *which assigns an arc expression to each arc* a *such that* $Type(E(a)) = C(p)$ *where* p *is the place connected to the arc* a*;*
- $I : P \to \mathrm{EXPR}_{\emptyset}$ *is an* initialization function *which assigns an initialization expression to each place* p *such that* $Type(I(p)) = C(p)$

where Var *is a function that returns the set of free variables in an expression,* $Type$ *is a function that returns the type, i.e., the color of a variable or the type returned by an expression and* EXPR_X *is the set of expressions provided by an inscription language, where the subscript* X *indicates the set of free variables that may occur in the expression.*

The net structure consists of a finite set P of places, a finite set T of transitions, and a finite set A of directed arcs, A. As in classical PNs, we require P and T to be disjoint. Each variable is required to have a type which belongs to 2^{Σ} as well as places, which, by means of the color set function $C : P \to 2^{\Sigma}$, are assigned to a set of colors belonging to 2^{Σ}. Also the type $Type(E(p,t))$ of arc expressions $E(p,t)$ where $p \in P$ and $t \in T$ is required to match the type $C(p)$ of the starting place, and analogously for an arc expression $E(t',p')$ where $t' \in T$ and $p' \in P$, $Type(E(t',p')) = C(p')$ must hold. Such restrictions imply that each place is marked with token of specific types only. The type of guard function $G : T \to \mathrm{EXPR}_V$ is instead required to be a boolean expression. Finally, the initialization function $I : P \to \mathrm{EXPR}_{\emptyset}$ assigns to each place p an initialization expression $I(p)$ (which is required to evaluate to a multiset over the color set of the place p). The initialization expression must be a closed expression.

We now move to the semantics of CPNs, i.e., how they evolve, by defining the enabling and occurrence of a step. The enabling rule specifies when a step (consisting of a multiset of binding elements, see later) is enabled in a given marking, and the occurrence rule specifies how the marking changes when an enabled step occurs.

Definition 5.4 (Marking)**.** *A* marking *is a function* M *that maps each place* $p \in P$ *into a multiset of tokens* $M(p) \in C(p)$*.*

The individual elements in the multiset $M(p)$ are called *tokens*. Of course, M is required to match the type of each place $p \in P$.

Given that guards and arc inscriptions have free variables, we need an assignment function to evaluate them. We denote by $Var(t)$ the set of free variables appearing in the guard and in any of the arc expressions on any arcs connected to a transition $t \in T$.

Definition 5.5 (Binding)**.** *A* binding *of a transition t is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type(v)$.*

The set of all bindings for a transition t is denoted by $B(t)$, and a binding element is a pair (t, b) such that $t \in T$ and $b \in B(t)$. The set of all binding elements $BE(t)$ for a transition t is defined by $BE(t) = \{(t, b) \mid b \in B(t)\}$. The set of all binding elements in a CPN schema is denoted by BE, and a *step* $Y \in BE$ is a non-empty finite multiset of binding elements. A CPN has a distinguish initial marking, obtained by evaluating the initialization expressions, which, having no free variables, can be evaluated with the empty binding []. The *initial marking* M_0 of a CPN is defined by $M_0(p) = I(p)[]$ for all $p \in P$.

We introduce the formal rules for enabling and occurrence of a steps by considering a single binding element only, and then generalize to steps with more binding elements.

The enabling and occurrence of a steps are based on evaluations of guards and arc expressions. Given a binding element (t, b) we denote by $G(t)[b]$ the result of evaluating the guard expression $G(t)$ of a transition t in the binding b. Analogously for $E(a)[b]$. Guard expressions evaluates to boolean, while the result of an arc expression is a multiset of token.

Definition 5.6 (Enabling and occurrence of a binding element)**.** *A binding element $(t, b) \in BE$ is enabled in a marking M if and only if the conjunction of following holds:*

1. *$G(t)[b]$ evaluates to true;*
2. *$\forall p \in P.E(p, t)[b] \ll= M(p)$.*

When (t, b) is enabled at M, it may occur, leading to the marking M' defined by:

3. *$\forall p \in P.M'(p) = (M(p) -- E(p, t)[b]) ++ E(t, p)[b]$.*

Intuitively, in order for a binding element to be enabled in a marking M, two conditions must hold: *(i)* the guard of the transition evaluates to true and *(ii)* there are enough tokens in the input places of the transition. The kind and number of token needed for a transition is specified by the expression $E(p, t)$. When an enabled binding element (t, b) occurs, it: *(i)* removes token from the input places of t (according to $E(p, t)[b]$) and *(ii)* adds the multiset of tokens in the output place (according to $E(p, t)[b]$). We denote by $M \xrightarrow{(b,t)} M'$ the occurrence of a binding element (b, t).

We now define the enabling and occurrence of steps.

Definition 5.7 (Enabling and occurrence of a step)**.** *A step $Y \in BE$ is* enabled *in a marking M if and only if the conjunction of the two following holds:*

1. $\forall (t,b) \in Y.G(t)[b]$ *evaluates to true;*
2. $\forall p \in P. \sum_{(t,b)\in Y} E(p,t)[b] \ll= M(p)$.

When Y *is enabled in* M*, it may* occur*, leading to the marking* M' *defined by:*

3. $\forall p \in P. M'(p) = (M(p) -- \sum_{(t,b)\in Y} E(p,t)[b]) ++ \sum_{(t,b)\in Y} E(t,p)[b]$.

As before, each binding element of a step must satisfy the guard of t. Furthermore, all binding elements in the step Y must be allowed to remove their own *private* tokens without sharing these tokens with other binding elements included in the step. This assumption is quite important, because it guarantees that the effect of the occurrence of a set of concurrent binding elements is the sum of the effects caused by the occurrence of the individual binding elements. This means that the marking reached will be the same as that which will be reached if the set of binding elements occur sequentially, i.e., one after another in some arbitrary order.

We denote a step Y occurring in M_1 leading to M_2 by $M_1 \xrightarrow{Y} M_2$.

Definition 5.8 (Finite occurrence sequence)**.** *A* finite occurrence sequence *of length* $n \geq 0$ *is an alternating sequence of markings and steps, written as:*

$$M_1 \xrightarrow{Y_1} M_2 \ldots M_n \xrightarrow{Y_n} M_{n+1}$$

such that $M_i \xrightarrow{Y_i} M_{i+1}$ *for all* $1 \leq i \leq n$*. All markings in the sequence are said to be* reachable *from* M_1*.*

Analogously, we can define an *infinite* sequence $M_1 \xrightarrow{Y_1} M_2 \ldots$ and the set of reachable markings is denoted by $\mathscr{R}(M_1)$.

5.2 Verification of CPN

CPN schemas can represent processes, the behavior of information systems or protocols. Given the complexity and high level of parallelism of such systems, their offline analysis, i.e., testing whether they satisfy specific *behavioral properties*, is often as important as the execution of the schema itself. The mathematical foundation of CPN allows for this kind of analysis: properties are expressed, as customary, in temporal logic, and model checking techniques are used to verify them. Indeed, once the precise semantics of steps has been formalized, all possible behaviors of the net, i.e., the set of possible markings, can be automatically computed.

This section illustrates the main techniques used to model check CPNs and shows that verification for CPN is in general undecidable, unless restrictions on both the structure of the net and the data manipulated are put in place.

The general procedure used in model checking to analyze the behavior of whatever system, program or protocol also applies in the context of CPNs: a transition system, called *state space* which represent all possible net's evolutions, is build, and it is composed by a node for each reachable marking and an arc for each occurring binding element. An arc labelled with a binding element (t,b) from a node representing a marking M_1 to a node representing a marking M_2 is present in the state space if and only if the binding (t,b) is enabled in M_1 and the occurrence of (t,b) in

M_1 leads to the marking M_2. Once such a TS is build, off-the-shelf model checking techniques can be used as a black box. Unfortunately, in general, the state space of the TS, i.e., the set of reachable markings $\mathscr{R}(M_1)$, is infinite. Intuitively, the unboundedness of the state space is due to two main reasons:

1. *unbounded places*: each place can contain an arbitrary number of tokens and
2. *unbounded colors*: places can have infinite color sets.

Intuitively, (1) says that the net is such that one place can cumulate an infinitely number of token and (2) that the values contained in the net can be infinitely many. Notice that even classical PN can have unbounded places and, for CPNs, that having bounded places does not guarantee a finite state space. In order to have a finite state space, we need both bounded places *and* bounded colors. From now on, we will assume the state space to be finite.

A number of different properties are of interest to be verified over the state space, but there are a few classes of properties which are used in the majority of practical cases. In what follows we present such classes.

Reachability properties express whether a marking M' is reachable (usually from the initial marking M_0). This is probably one of the most interesting properties in computer science: to give an example, a planning problem in AI can be reduced to synthesis of a reachability property (see Section 1.3). In LTL they can be expressed as $\boldsymbol{F}\phi$, where ϕ is a formula that holds only in marking M'. In μ-calculus it can be asserted as $\mu Z.\phi \vee \boldsymbol{X}Z$. To verify such a property it is enough to find a path in the state space which eventually reaches M' starting from the initial state. Technically, this is a least fixpoint computation.

A more CPN-specific class of properties are the *boundedness properties*, which specify how many tokens a place can hold, when all reachable markings are considered. This number is called *best upper integer bound* of a place. Analogously, the *best lower integer bound* specifies the minimal number of tokens that can reside on a specific place in any reachable marking. Expressing these properties in a temporal logic is involved, because they are not boolean and require second-order predicates.

Liveness properties describe the requirement that the system makes progresses toward a specific goal. In LTL they are expressed as $\boldsymbol{G}(\psi \rightarrow \boldsymbol{F}\phi)$ where ϕ is the goal. In CPN it is usually of interest to check whether a way to progress in the net always exists, i.e., there are no *dead markings* where no binding elements are enabled, or, if there are, some property holds, e.g., the result is correct.

Fairness properties are used to check whether the system is "fair" or if it shows some infinite undesired behavior. In LTL such properties are usually expressed as $\boldsymbol{GF}\phi \rightarrow \boldsymbol{GF}\psi$. In CPN, they are used to check how often transitions occur in infinite occurrence sequences. A transition t is called *impartial* if it occurs infinitely often in all infinite occurrence sequences. This implies that the removal of t or blocking it by means of a guard $false$ will remove all infinite occurrence sequences from the CPN schema. The impartiality property can be generalised to binding elements, sets of binding elements, and sets of transitions.

Lastly, we consider *home properties*, that can be considered a special kind of fairness properties. A home property tells us if there exists a single *home marking* M_h which can be reached from any reachable marking. In other words, this means that it is always possible to extend an occurrence sequence to reach M_h. A home property can be easily expressed in LTL as $\exists M.\boldsymbol{GF}M$.

When modeling a system with CPNs, it is very easy to get an infinite state space. In order to perform the aforementioned analysis it is necessary to bound the state space, usually by restricting to a specific configuration of the system parameters, e.g., a hundred token in all places and bounded data values. When restriction are adopted, the result of the analysis is not exhaustive. To increase the quality of the analysis what can be done is incrementally relaxing the constraints, e.g., considering more token and data values. As the value of parameters increases, the state space grows rapidly. This issue is known as the *state explosion problem*, and has been addressed largely in model checking literature, and several *reduction methods* for the state space has been proposed.

One class of methods aims at exploring a subset of the state space only. Since the search is not exhaustive, if an error is found, meaning that the property has been falsified, then the system has a real bug, but if the property is verified, then the system may or may not be correct. This because in general the correctness of the partial state space does not imply that the remaining part of the state space is correct as well. In order to increase the correctness of the verification process such techniques either use heuristics or they exploit symmetries of the system or use compact representation for the state explored in order to increase the correctness of the verification (e.g., ample-sets, bounded model checking, bit state hashing).

Another approach is to delete states from the memory during state space exploration. These methods can explore the full state space and also consider one state more than once (sweep line).

Some methods computes a condensed (or abstract) state space, where each node represents a set of concrete states and is, indeed, an equivalence class.

Many of the aforementioned techniques suffer from false positives, i.e., when they return that the property is verified, such a result could be incorrect, because the correctness of the reduced state space does not entails the correctness of the complete state space.

What is important about the CPN state space analysis is that, even if the reduction methods were very efficient, an exhaustive analysis of the whole state space would be impossible, because the modeling power of CPN makes verification tasks undecidable (see, e.g., [64] for a survey). Moreover, it is difficult to find syntactic properties ensuring the finiteness of the state space.

We think artifact-centric systems are a good compromise between expressive power and decidability of verification. Besides we provided a syntactic condition that guarantees the finiteness of the execution transition system, i.e., of the state space. In the next Section we show that DCDSs are expressive enough to capture CPNs.

5.3 Capturing CPNs with DCDS

We study the following problem: given a CPN C and an initial marking M_0, can we capture the semantics of CPN by means of a DCDS $\mathcal{S}$? In general, the answer to this question is negative, and the purpose of this section is not addressing this problem formally, but rather to analyze the main intuitions behind it. To be more precise, the semantics of a CPN is given in terms of enabling and occurrence of a step and, hence, in terms of a transition system that represents the possible evolutions of the CPN. Such a transition system Υ_C can be defined as follows: we start from an initial marking M_0 and, for each possible set of steps $\boldsymbol{Y}_i = \{Y_{t_1} \dots Y_{t_k}\}$, where each Y_{t_j} is an enabled step for a transition t_j, we make $\boldsymbol{Y}_i$ occur, obtaining M_{i+1}. We keep going on by repeating the same procedure to all new generated markings in any possible way. Such a possible infinite Υ_C actually captures the behavior of a CPN, because the semantics of a CPN allows for: *(i)* firing more than one binding for transition, i.e., a step (provided that the bindings in the step are independent) and *(ii)* more steps concurrently, if they refer to different transitions.

Our problem can hence be rephrased as follows: Given a CPN C, an initial marking M_0, and its transition system Υ_C, does a DCDS $\mathcal{S} = (\mathcal{D}, \mathcal{P})$ with $\mathcal{D} = (\mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0)$ and $\mathcal{P} = (\mathcal{F}, \mathcal{A}, \varrho)$ such that $\Upsilon_{\mathcal{S}} \equiv \Upsilon_C$ exist?

For the sake of simplicity, we assume the CPN to not have multisets, for we have a classic relational settings. This assumption can be easily dropped when using relational algebra supporting multisets, like in [77, 91]. Definitions is Section 5.1 are general enough to cover the case in which each place contains at most one token of a given set of colors.

Let us focus first on the syntax by considering whether we have the necessary data structures for statically represent a CPN and a marking. A first order, i.e., relational, structure is sufficient for the purpose. More precisely, the schema $\mathcal{R}$ contains a relation $R_p(A_1 \dots A_n)$ for each place $p \in P$ where $A_1 \dots A_n$ are the colors assigned to p through function C. It is easy to see that an instance $\mathcal{I}$ of $\mathcal{R}$ represent exactly a marking M for C.

We now move to the semantic aspects, i.e., we analyze whether the dynamic behavior of C can be simulated by a DCDS. In Section 5.1 we did not enter into details of the inscription language, but from now on we assume that it can be expressed in first order logic. We start by considering the enabling and occurrence of a binding. Given a binding element (b, t) and markings M, M' such that $M \xrightarrow{(b,t)} M'$, can we model the occurrence of (b, t) in a DCDS? Let us assume to have the relational instance I that represents M, then we make use of action $\alpha_t(a_1, \dots, a_m) : \{e_1\}$ to simulate the occurrence of (b, t). Parameters $a_1 \dots a_m$ will be substituted with values given by b, and the effects will generate the new instance $\mathcal{I}'$ representing marking M'. The guard $G(t)$ and conditions for enabling transition t (given by arc inscriptions) are translated in a FO formula and will be part of effects and the precondition of the action.

It is easier to illustrate the translation with the help of the example in Figure 5.1. Transition t has two input places p_1 and p_2 and two output places p_3 and p_4.

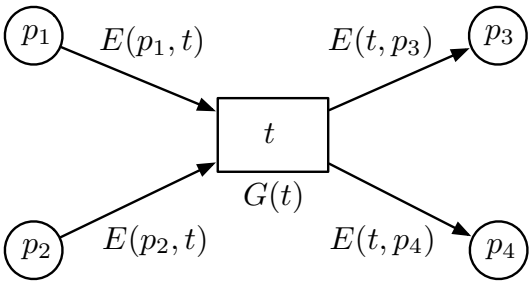


Figure 5.1. Example of a transition in a CPN.

The guard is

$$G(t) := (1`\,\mathsf{NewYork} +\!\!+ 1`\,\mathsf{Rome}) \ll\!= \; M(p_3)$$

while the arc inscriptions are:

- $E(p_1, t) := 1`\,(s, 0)$;
- $E(p_2, t) := 1`\,c$;
- $E(t, p_3) := if\ c = \mathsf{NewYork}\ then\ 1`\,(s, n)\ else\ \emptyset$
- $E(t, p_4) := if\ c = \mathsf{Rome}\ then\ 1`\,(s, n)\ else\ \emptyset$.

where s, n and c are variables, Rome and NewYork are constants and with $x`\,token$ we mean that x is the number of appearances of $token$ in the multiset. We are assuming the number of appearances to be 1 for every token.

The initial marking is:

- $M(p_1) := \{1`\,(\mathsf{msg1}, 0), 1`\,(\mathsf{msg5}, 0)\}$;
- $M(p_2) := \{1`\,\mathsf{Rome}, 1`\,\mathsf{NewYork}\}$;
- $M(p_3) := \{1`\,(\mathsf{msg3}, 12)\}$;
- $M(p_4) := \emptyset$.

Let us assume to have the binding (b, t) where:

- $b(s) := \mathsf{msg1}$;
- $b(n) := 22$;
- $b(c) := \mathsf{Rome}$.

Binding (b, t) is enabled in M, since guard $G(t)$ is satisfied as well as conditions on arc inscriptions. The occurrence of (b, t) leads to marking M':

- $M'(p_1) := \{1`\,(\mathsf{msg5}, 0)\}$;
- $M'(p_2) := \{1`\,\mathsf{NewYork}\}$;
- $M'(p_3) := \{1`\,(\mathsf{msg3}, 12)\}$;
- $M'(p_4) := \{1`\,(\mathsf{msg1}, 22)\}$.

We can now show the translation in a DCDS. The schema is $\mathcal{R} = \{R_{p_1}/2, R_{p_2}/1, R_{p_3}/2, R_{p_4}/2\}$ where each relational symbol R_{p_i}/n has arity n and will store the marking for place p_i. The initial instance $\mathcal{I}_M$ is:

$$\begin{aligned}\mathcal{I}_M := \{&R_{p_1}(\mathsf{msg1}, 0), R_{p_1}(\mathsf{msg5}, 0)\\ &R_{p_2}(\mathsf{Rome}), R_{p_2}(\mathsf{NewYork})\\ &R_{p_3}(\mathsf{msg3}, 12)\}\end{aligned}$$

Next, we show how to model transition t in C as an action α_t in $\mathcal{S}$. Bindings in CPN provide a mechanism to possibly add values from the external world into the system. In our example, the number 22 associated to msg1 represents the payload of msg1 which comes from the external world. In a DCDS input values are modeled as Skolem functions: this choice fits exactly the connotation of functions as call to external services. The other arc inscriptions variables, namely s and c, are modeled as parameters of $\alpha_t(s, c)$. Notice that such values are already present in the current instance. Hence precondition π for α_t is a formula with s and c free variables and it will encode the condition for enabling the binding, i.e., condition 1 and 2 of Definition 5.6 as follows:

$$\begin{aligned}&(1)\ \ R_{p_2}(\mathsf{Rome}) \wedge R_{p_2}(\mathsf{NewYork}) \wedge\\ &(2)\ \ R_{p_1}(s, 0) \wedge\\ &(3)\ \ R_{p_2}(c) \mapsto \alpha_t(s, c)\end{aligned}$$

where (1) is the formula that checks if the guard $G(t)$ is satisfied, analogously as condition 1 of Definition 5.6. Formula (2) and (3) check condition 2 of Definition 5.6 for arc inscription $E(p_1, t)$ and $E(p_2, t)$ respectively. External input n is modeled as a function $f(\cdot)$ in the action effects.

The action fires only if the precondition π is satisfied, and the effects have to: *(i)* remove tokens $(s, 0)$ and c from places p_1 and p_2 respectively and *(ii)* add token $(s, f(\cdot))$ in p_3 if $c = \mathsf{NewYork}$ or add token $(s, f(\cdot))$ in p_4 if $c = \mathsf{Rome}$. Results *(i)* are carried out by the following effects:

1. $R_{p_1}(y, z) \wedge y \neq s \wedge z \neq n \rightsquigarrow R_{p_1}(y, z)$ that copies all tokens in place p_1 but the one being consumed;
2. $R_{p_2}(y) \wedge y \neq c \rightsquigarrow R_{p_2}(y)$ that copies all tokens in place p_2 but the one being consumed.

We now have to update tokens in places p_3 and p_4 in the following way: if $c = \mathsf{NewYork}$, then $(s, f(\cdot))$ should be added to p_3, while if $c = \mathsf{Rome}$, then it should be moved to p_4. Let us first consider the first case, i.e., $c = \mathsf{NewYork}$, then we have to add token $(s, f(\cdot))$ in place p_3. Analogously for $c = \mathsf{Rome}$ and place p_4. Each of those cases is handled by a different effect, as follows:

3. $c = \mathsf{NewYork} \wedge R_{p_3}(y, z) \rightsquigarrow R_{p_3}(y, z) \wedge R_{p_3}(s, f(\cdot))$ copies all tokens in p_3, if any, plus $(s, f(\cdot))$ when $c = \mathsf{NewYork}$;
4. $c \neq \mathsf{NewYork} \wedge R_{p_3}(y, z) \rightsquigarrow R_{p_3}(y, z)$ just copies all tokens in p_3 because, if $c \neq \mathsf{NewYork}$, no new token have to be added;

5. $c = \mathsf{Rome} \wedge R_{p_4}(y, z) \rightsquigarrow R_{p_4}(y, z) \wedge R_{p_4}(s, f(\cdot))$ is analogous to effect 3;
6. $c \neq \mathsf{Rome} \wedge R_{p_4}(y, z) \rightsquigarrow R_{p_4}(y, z)$ is analogous to effect 4.

If we fire action α_t from instance $\mathcal{I}_M$ with parameters substituted with values of the binding, i.e., we perform $\text{DO}(\mathcal{I}_M, \alpha, \{s/\mathsf{msg1}, c/\mathsf{Rome}\})$ we obtain the following instance $\overline{\mathcal{I}}_{M'}$:

$$\begin{aligned}\overline{\mathcal{I}}_{M'} := \{&R_{p_1}(\mathsf{msg5}, 3)\\ &R_{p_2}(R_{p_2}(3, \mathsf{NewYork})\\ &R_{p_3}(\mathsf{msg3}, 12)\\ &R_{p_4}(\mathsf{msg1}, f(\cdot))\}.\end{aligned}$$

Notice that function $f(\cdot)$ is still undefined. However, among the evaluations of $f(\cdot)$ there is a function $\theta_{22} \in \text{EVAL}_{\mathcal{S}}(\mathcal{I}_M, \alpha, \{s/\mathsf{msg1}, c/\mathsf{Rome}\})$ that evaluates $f(\cdot)$ to 22. As a result, there is a instance generated from $\mathcal{I}_M$:

$$\begin{aligned}\text{DO}(\mathcal{I}_M, \alpha, \{s/\mathsf{msg1}, c/\mathsf{Rome}\})\theta_{22} := \mathcal{I}_{M'} := \{&R_{p_1}(\mathsf{msg5}, 3)\\ &R_{p_2}(R_{p_2}(3, \mathsf{NewYork})\\ &R_{p_3}(\mathsf{msg3}, 12)\\ &R_{p_4}(\mathsf{msg1}, 22)\}.\end{aligned}$$

that corresponds exactly to marking M'.

In general, we have an action α_t for each transition t of a CPN. The guard $G(t)$ is encoded in the precondition π of α_t, as well as all arc inscriptions from the input places to the transition t. Arc inscriptions from t to the output places are instead encoded in the action affects.

We now explain the main intuitions of such a translation. While it is intuitive why we encoded the guard of a transition as part of the precondition of the action, the reason behind the encoding of the arc inscription is more subtle. The purpose of arcs from input places is twofold: *(i)* they check whether there are enough token to fire the transition, so, intuitively, they represent a precondition of the action and *(ii)* they "consume" tokens from output places according to Definition 5.6. We encode the precondition role in *(i)* as a precondition of action α_t, and the role in *(ii)* as an effect. Concerning the arc inscriptions to output places, their meaning is to add tokens to output places, and hence they are encoded in the effects as well. Possibly external inputs are modeled as external service calls.

Binding values occur both in the input arc inscriptions and output ones, while in our framework service call functions are allowed in the right-hand part of effects only. However, considered that the variables of an incoming arc inscription have to be the set of colors of the input place, all bindings with values that do not appear in the place cannot be satisfied. It is hence formally correct to use new values in the right-part of the effects only. Besides, from the semantics in Definition 5.6, we can easily assume that the translation in DCDS of incoming arc inscriptions to be domain independent, and, hence at each step we have a finite set of answers.

As a result of these considerations, it is possible to capture the semantics of occurrence of a binding element in a DCDS. Can we also simulate the transition

system Υ_C? We conjecture a negative answer to this question, unless C is finite. Indeed, there is no easy way to fire in a DCDS all possible set of steps $\boldsymbol{Y}_i$ of C. Recall that $\boldsymbol{Y}_i = \{Y_{t_1} \dots Y_{t_k}\}$ where each Y_{t_j} is a step for transition t_j that, in turn, is a set of *independent* binding elements (b, t_j). To obtain the same behavior from a DCDS, we should include preconditions that are able to select sets of independent assignments, but, unfortunately, our formalism is not expressive enough.

However, we are interested in analyzing a CPN (or a DCDS) for verification purposes, that are performed, as explained in Section 5.2, over the *state space*, that is quite different from Υ_C. Indeed, by definition, it is build by firing a binding element at a time only. The independence of bindings in a step guarantees that the marking reached by performing a step will be the same as that which will be reached if the set of binding elements in that step occur sequentially, i.e., one after another in some arbitrary order. From this we can infer that the set of markings reached by Υ_C is contained (or equals) in the set of markings of the state space $\mathscr{R}(M_1)$. Nonetheless, it is important to stress that the state space does not represent the possible evolutions of a CPN, and hence some temporal properties that are true in the state space may be not in the real execution of a CPN and vice-versa. For properties we mentioned in Section 5.2 we are guaranteed that if they hold in the state space, they do in Υ_C as well.

If we have a schema and actions as we described before, preconditions of the actions guarantees, by its semantics, that, given an instance I_M, we have a successor for each action, for each set possible substitution to parameters of the action and for each evaluation θ to the Skolem functions. Given a CPN C and an initial marking M_0 we can actually simulate the space state of a CPN with the execution transition system $\Upsilon_{\mathcal{S}}$ of the corresponding DCDS.

5.4 Discussion

We informally described how to capture a CPN in a DCDS. The scope of this Chapter is to show that the semantics of our system is powerful enough to capture the transition relation of CPNs. CPNs are used as a underlying formalism for several BPM languages, then in principle our framework could be used to serve the same purpose.

There is another important aspect of the reduction. Given that the work in [11] shows strong decidability results for verification of data-aware systems, the same results can be carried to CPNs. Verification of CPNs amounts to properties illustrated in Section 5.2 when the CPN is known to be bounded. When unbounded, existing model checking techniques that cope with state explosion problem could be used, such as bounded model checking, infinite-state model checking, or predicate abstraction, but all of those suffer from the issues explained in Section 5.2. Translating a CPN into a DCDS instead, brings several advantages: first of all the behavior of a DCDS is finitely abstracted, guaranteeing sound and complete verification procedures. Furthermore, the verification language is μ-calculus, that allows for more expressive power than the few properties presented in Section 5.2. Lastly, while it is easy to check when a DCDS model is decidable (it is sufficient to check whether the acyclicity property is met) it is not as much easy to tailor a fine-grained

syntactic property for CPNs that guarantees decidability of verification. In [84] semantic properties are illustrated, such as boundedness of places, but no syntactic properties are given. Coarse solutions ensuring boundedness consist in constraining arc inscriptions in order to add no new values in the net and generate no new tokens, but such way out basically downgrades a CPN to a classical PN, loosing any benefit of colors. The significance of results in [11] lies, instead, in a good trade-off between the use of unbounded data and the expressive power of the verification formalism. For verification purposes it is therefore reasonable and relevant to reduce a CPN to a DCDS.

As future works, it would be certainly of interest to understand how syntactic conditions of acyclicity of DCDSs could be translated in syntactic properties of CPNs, so as to exploit results in [11] directly over CPNs, without performing the translation.

Chapter 6

Knowledge and Action Bases

While the frameworks presented in Chapter 2 and 3 maintain information in a relational database, for more sophisticated applications it is foreseen to enrich data-intensive business processes with a semantic level, where information can be maintained in a semantically rich knowledge base that allows for operating with incomplete information [32, 94]. This leads us to look into how to combine first-order data, ontologies and processes, whilst maintaining basic inference tasks (specifically verification) decidable.

In this setting, we capture the domain of interest in terms of semantically rich formalisms as those provided by ontological languages based on Description Logics (DLs) [6]. Such languages natively deal with incomplete knowledge in the modeled domain. This additional flexibility comes with an added cost, however: differently from relational databases, to evaluate queries we need to resort to logical implication. Moreover incomplete information combined with the ability of evolving the system through actions results in a notoriously fragile setting w.r.t. decidability [139, 70]. To overcome this difficulty, virtually all solutions that aim at robustness are based on a so-called "functional view of knowledge bases" [93]: the KB provides the ability of querying based on logical implication ("ask"), and the ability of progressing it to a "new" KB through forms of updates ("tell") [7, 34]. We follow this functional view of KBs. However, a key point of our work is that at each execution step external information is incorporated into the system in form of new individuals (denoted by *Skolem terms*), that is, our systems are not closed w.r.t. the available information. This makes our framework particularly interesting and challenging. In particular, the presence of these individuals requires a specific treatment of equality, since as the system progresses and new information is acquired, distinct Skolem terms may be inferred to denote the same object.

Specifically, we introduce the so-called *Knowledge and Action Bases* (KABs). A KAB is equipped with an ontology, or more precisely, a TBox, expressed, in our case, in a variant of *DL-Lite*$_{\mathcal{A}}$ [31], which extends the core of the Web Ontology Language OWL 2 QL (`http://www.w3.org/TR/owl2-profiles/`) and is particularly well suited for data management. Such a TBox captures intensional information on the domain of interest, similarly to UML class diagrams or other conceptual data models, though as a software component to be used at runtime.

The KAB includes also an ABox, which acts as a storage or state by maintaining

the data of interest, which are accessed by relying on query answering based on logical implication (certain answers). Notably, our variant of *DL-Lite*$_{\mathcal{A}}$ is without UNA, and we allow for explicit equality assertions in the ABox. In this way, we can suitably treat Skolem terms to represent individuals acquired during the execution. Technically, the need of dealing with equality breaks the first-order rewritability of *DL-Lite*$_{\mathcal{A}}$ query answering, and requires that, in addition to the rewriting process needed for query answering, inference on equality is performed [5].

As a query language, we use unions of conjunctive queries, possibly composing their certain answers through full FOL constructs. This gives rise to an *epistemic query language* that asks about what is "known" by the current KB [30].

Apart from the KB, the KAB follows the spirit of Chapter 2 and 3 and hence it contains *actions*, specified as sets of conditional effects whose execution changes the state of the ABox, and a *process*, modeled as condition/action rules, that specifies which actions can be executed at each step. Both conditions of actions and conditions of process' rules are specified as epistemic queries over the KB, and therefore they are interpreted under the certain answer semantics.

In this setting, we address the verification of temporal/dynamic properties expressed in a first-order variant of μ-calculus [113, 125], where atomic formulae are queries over the KB that can refer both to constants and Skolem terms, and where a controlled form of quantification across states is allowed. Notice that all previous decidability results on actions over DL KBs assumed that no information is coming from outside of the system, in the sense that no new individual terms are added while executing actions [34, 7, 120]. Here, instead, we allow for arbitrary introduction of new terms.

Unsurprisingly, we show that even for very simple KABs and temporal properties, verification is undecidable. However, we also show that for a rich class of KABs, verification is in fact decidable and reducible to finite-state model checking. To obtain this result, following the intuitions of Chapter 2 and 3, we rely on recent results in data exchange on the finiteness of the chase of tuple-generating dependencies [65], though, in our case, we need to extend the approach to deal with *(i)* incomplete information, *(ii)* inference on equality, and *(iii)* quantification across states in the verification language.

6.1 Knowledge base formalism

We assume some familiarity with Description Logics and refer to [6] for the basic notions. Here, we just introduce the terms and notations that are necessary for the understanding of the technical development of the chapter.

For expressing knowledge bases, we use *DL-Lite*$_{\text{NU}}$, a variant of the *DL-Lite*$_{\mathcal{A}}$ language [116, 29] in which we drop the *unique name assumption* (UNA) [5]. The syntax of *concept* and role *expressions* in *DL-Lite*$_{\text{NU}}$ is as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl} B & \longrightarrow & N \mid \exists R & R & \longrightarrow & P \mid P^- \\ C & \longrightarrow & B \mid \neg B & V & \longrightarrow & R \mid \neg R \end{array}$$

where N denotes a *concept name*, P a *role name*, and P^- an *inverse role*. A *DL-Lite*$_{\text{NU}}$ *knowledge base* (KB) is a pair (T, A), where:

- T is a TBox, i.e., a finite set of *TBox assertions* of the form

$$B \sqsubseteq C, \qquad R \sqsubseteq V, \qquad (\mathsf{funct}\ R),$$

  called respectively *concept inclusions*, *role inclusions*, and *functionality assertions*.

- A is an Abox, i.e., a finite set of *ABox assertions* of the form

$$N(t_1), \qquad P(t_1, t_2), \qquad t_1 = t_2,$$

  called respectively, *concept (membership) assertions*, *role (membership) assertions*, and *equality assertions*, where t_1, t_2 are constants, or more generally terms (see later) denoting individuals.

As usual in *DL-Lite*, a TBox may contain neither $(\mathsf{funct}\ P)$ nor $(\mathsf{funct}\ P^-)$ if it contains $R \sqsubseteq P$ or $R \sqsubseteq P^-$, for some role R.

We adopt the standard semantics of DLs based on FOL interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the interpretation domain and $\cdot^{\mathcal{I}}$ is the interpretation function such that $c^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, $N^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, and $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The semantics of the constructs, of TBox and ABox assertions, and the notions of *satisfaction* and of *model* are as usual. We also say that A is *consistent w.r.t.* T if (T, A) is satisfiable, i.e., admits at least one model.

Next we introduce queries. As usual (cf. OWL 2), answers to queries are formed by constants/terms denoting individuals explicitly mentioned in the ABox. The *domain of an ABox* A, denoted by $\textsc{adom}(A)$, is the (finite) set of constants/terms appearing in concept, role, and equality assertions in A. The (predicate) alphabet of a KB (T, A), denoted $\textsc{alph}((T, A))$ is the set of concept and role names occurring in $T \cup A$.

A *union of conjunctive queries* (UCQ) q over a KB (T, A) is a FOL formula of the form $\exists \vec{y}_1 . conj_1(\vec{x}, \vec{y}_1) \vee \cdots \vee \exists \vec{y}_n . conj_n(\vec{x}, \vec{y}_n)$, with free variables $\vec{x}$ and existentially quantified variables $\vec{y}_1, \ldots, \vec{y}_n$. Each $conj_i(\vec{x}, \vec{y}_i)$ in q is a conjunction of atoms of the form $N(z)$, $P(z, z')$, where N and P respectively denote a concept and a role name occurring in $\textsc{alph}((T, A))$, and z, z' are constants in $\textsc{adom}(A)$ or variables in $\vec{x}$ or $\vec{y}_i$, for some $i \in \{1, \ldots, n\}$. The *certain answers* to q over (T, A) is the set $\textsc{ans}(q, T, A)$ of substitutions (as customary, we can view each substitution simply as a tuple of constants, assuming some ordering of the free variables of q) σ of the free variables of q with constants/terms in $\textsc{adom}(A)$ such that $q\sigma$ is logically implied by (T, A), written, $T, A \models q\sigma$, i.e., $q\sigma$ evaluates to true in every model of (T, A). If q has no free variables, then it is called *boolean* and its certain answers are either the empty substitution denoting `true` or nothing denoting `false`.

Theorem 6.1 ([5]). *Computing* $\textsc{ans}(q, T, A)$ *of an UCQ* q *over KBs* (T, A) *is* PTIME*-complete in the size of* T *and* A.

We also consider an extension of UCQs, called ECQs, which are queries of the query language *EQL-Lite*(UCQ) [30], that is, the FOL query language whose atoms

are UCQs evaluated according to the certain answer semantics above. An *ECQ* over a KB (T, A) is a possibly open formula of the form:

$$Q \longrightarrow [q] \mid [x = y] \mid \neg Q \mid Q_1 \wedge Q_2 \mid \exists x.Q$$

where $[q]$ denotes the certain answers of a UCQ q over (T, A), $[x = y]$ denotes the certain answers of $x = y$ over (T, A), that is the set $\{\langle x, y\rangle \in \textsc{adom}(A) \mid T, A \models (x = y)\}$, logical operators have the usual meaning, and quantification ranges over elements of $\textsc{adom}(A)$.

Formally we define the relation Q *holds in* (T, A) *under substitution* σ, written $T, A, \sigma \models Q$, as follows:

$T, A, \sigma \models [q]$	if	$T, A \models q\sigma$
$T, A, \sigma \models [x = y]$	if	$T, A \models (x = y)\sigma$
$T, A, \sigma \models \neg Q$	if	$T, A, \sigma \not\models Q$
$T, A, \sigma \models Q_1 \wedge Q_2$	if	$T, A, \sigma \models Q_1$ and $T, A, \sigma \models Q_2$
$T, A, \sigma \models \exists x.Q$	if	exists $t \in \textsc{adom}(A)$ such that $T, A, \sigma[x/t] \models Q$

where $\sigma[x/t]$ denotes the substitution obtained from σ by assigning to x the constant/term t (if x is already present in σ its value is replaced by t, if not, the pair x/t is added to the substitution).

The *certain answer to* Q *over* (T, A), is the set of substitutions σ for the free variables in Q such that $T, A, \sigma \models Q$, i.e.,

$$\textsc{ans}(Q, T, A) = \{\sigma \mid T, A, \sigma \models Q\}$$

Following the line of the proof in [30], but considering Theorem 6.1 for the basic step of evaluating an UCQ, we get:

Theorem 6.2. *Computing* $\textsc{ans}(Q, T, A)$ *of an ECQ* Q *over KBs* (T, A) *is* PTIME*-complete in the size of* T *and* A*.*

We close by recalling that DL-Lite enjoys a *rewritability* property, which in our setting says that for every UCQ q,

$$\textsc{ans}(q, T, A) = \textsc{ans}(\mathit{rew}(q), \emptyset, A),$$

where $\mathit{rew}(q)$ is a UCQ computed by the reformulation algorithm in [31]. Notice that, in this way, we have "compiled away" the TBox, though we still need to do logical implication w.r.t. the ABox, which contains equality assertions. This result can be extended to ECQs as well, i.e., for every ECQ Q, $\textsc{ans}(Q, T, A) = \textsc{ans}(\mathit{rew}(Q), \emptyset, A)$ where the query $\mathit{rew}(Q)$ is obtained from Q by substituting each atom $[q]$ (where q is an UCQ) by $[\mathit{rew}(q)]$ [30].

We say that two ABoxes A_1 and A_2 are equivalent w.r.t. TBox T and predicate alphabet Λ, denoted by

$$A_1 \equiv_{T,\Lambda} A_2$$

if for every ABox assertion $\alpha_2 \in A_2$ which is either a concept assertion $N(t)$ with $N \in \Lambda$, role assertion $P(t_1, t_2)$ with $P \in \Lambda$, or equivalence assertion $t_1 = t_2$, we have $T, A_1 \models \alpha_2$; and vice-versa, for every ABox assertion $\alpha_1 \in A_1$, which is either

a concept assertion $N(t)$ with $N \in \Lambda$, role assertion $P(t_1, t_2)$ with $P \in \Lambda$, or equivalence assertion $t_1 = t_2$, we have $T, A_2 \models \alpha_1$. Notice that if $A_1 \equiv_{T,\Lambda} A_2$, then for every ECQ Q whose concept and role names belong to Λ we have that $\text{ANS}(Q, T, A_1) = \text{ANS}(Q, T, A_2)$. Notice also that, by applying Theorem 6.2 to the boolean query $[\alpha]$ corresponding to the ABox assertion α, for each α in A_1 and A_2, we obtain that ABox equivalence can be checked in PTIME.

6.2 The framework

A *Knowledge and Action Base (KAB)* is a tuple $\mathcal{K} = (\mathcal{T}, A_0, \Gamma, \Pi)$ where T and A_0 form the *knowledge component* (or knowledge base), and Γ and Π form the *action component* (or action base). In practice, $\mathcal{K}$ is a stateful device that stores the information of interest into a KB, formed by a fixed TBox $\mathcal{T}$ and an initial ABox A_0, which evolves by executing actions Γ according to the sequencing established by process Π. During the evolution new individuals can be acquired by the KB. Such individuals are witnesses of new pieces of information inserted into the KAB from the environment the KAB runs in (i.e., the external world). We represent these new objects as Skolem terms. As the KAB evolves the identity of individuals should be intuitively preserved and this induces the necessity of remembering equalities between terms denoting individuals discovered in the past. We describe in detail the components of the KAB.

TBox. $\mathcal{T}$ is a *DL-Lite*$_{\text{NU}}$ TBox, used to capture the intensional knowledge about the domain of interest. Such a TBox is fixed once and for all, and does not evolve during the execution of the KAB.

ABox. A_0 is a *DL-Lite*$_{\text{NU}}$ ABox, which stores the extensional information of interest. Notice that A_0 is the ABox of the *initial state* of the KAB, and as the KAB evolves due to the effect of actions, the ABox, which is indeed the state of the system, evolves accordingly to store up-to-date information. Through actions, we acquire new information from the external world, which results in new individuals. These individuals are denoted by (ground) *Skolem terms*. The presence of Skolem terms has an impact on the treatment of equality, since in principle we need to close equality w.r.t. congruence, i.e., if $a = b$ holds, then also $f(a) = f(b)$ must hold. Closure w.r.t. congruence generates an infinite number of logically implied equality assertions. However, we are going to keep such assertions implicit, computing them only when needed. Observe that, given two complex terms, verifying their equality requires a PTIME computation.

Actions. Γ is a finite set actions. An *action* $\gamma \in \Gamma$ modifies the current ABox A by adding or deleting assertions, thus generating a new ABox A'. Action γ is constituted by a *signature* and an *effect specification*. The action signature is constituted by a name and a list of individual *input parameters*. Such parameters need to be instantiated with actual individuals at execution time. Given a substitution θ for the input parameters, we denote by $\gamma\theta$ the instantiated action with the *actual* parameters coming from θ. We disregard a specific treatment of output parameters,

and assume instead that the user can freely pose queries over the KB, extracting whatever information she/he is interested in. The *effect specification* of an action γ consists of a set $\{e_1, \ldots, e_n\}$ of effects, which take place simultaneously. An *effect* e_i has the form

$$[q_i^+] \wedge Q_i^- \rightsquigarrow A'_i, \tag{6.1}$$

where

- q_i^+ is an UCQ, i.e., a positive query, which extract the rough data to process (obtained as the certain answers of q_i^+); the free variables of q_i^+ include the action parameters;
- Q_i^- is an arbitrary ECQ, whose free variables occur all among the free variables of q_i^+, which refines, by using negation and quantification, the result of q_i^+. The query $[q_i^+] \wedge Q_i^-$ as a whole extracts individual terms to be used to form the new state of the KAB (notice that the UCQ-ECQ division is also a convenience to have readily available the positive part of the condition, which we will exploit later);
- A'_i is a set of (non-ground) ABox assertions, which include as terms: constants in A_0, free variables of q_i^+, and Skolem terms $f(\vec{x})$ having as arguments $\vec{x}$ free variables of q_i^+. These terms once grounded with the values extracted from $[q_i^+] \wedge Q_i^-$ give rise to (ground) ABox assertions, which contribute to form the next state of the KAB.

More precisely, given the current ABox A of $\mathcal{K}$ and a substitution θ for the parameters of the action γ, the new state A' resulting from firing the instantiated action $\gamma\theta$ on the state A, is computed as follows:

- Each effect $e_i \in \gamma$ of form (6.1) extracts from A the set $\textsc{ans}(([q_i^+] \wedge Q_i^-)\theta, T, A)$ of tuples of terms in $\textsc{adom}(A)$, and for each such tuple σ asserts a set $A'_i\theta\sigma$ of ABox assertions obtained from $A'_i\theta$ by applying the substitution σ for the free variables of q_i^+. For each Skolem term $f(\vec{x})\theta$ appearing in $A'_i\theta$, a new ground term is introduced having the form $f(\vec{x})\theta\sigma$. These terms represent new "constants" coming from the external environment the KAB is running in. We denote by $e_i\theta(A)$ the overall set of ABox assertions, i.e.,

$$e_i\theta(A) = \bigcup_{\sigma \in \textsc{ans}(([q_i^+] \wedge Q_i^-)\theta, T, A)} A'_i\theta\sigma.$$

- Moreover, let $\textsc{eq}(\mathcal{T}, A) = \{t_1 = t_2 \mid \langle t_1, t_2 \rangle \in \textsc{ans}([x_1 = x_2], T, A)\}$. Observe that, due to the semantics of queries, the terms in $\textsc{eq}(\mathcal{T}, A)$ must appear explicitly in $\textsc{adom}(A)$, that is, the possibly infinite number of equalities due to congruence do not appear in $\textsc{eq}(\mathcal{T}, A)$, though they are logically implied.

The overall *effect* of the action γ with parameter substitution θ over A is the new ABox $A' = do(T, A, \gamma\theta)$ where

$$do(T, A, \gamma\theta) = \textsc{eq}(\mathcal{T}, A) \cup \bigcup_{1 \leq i \leq n} e_i\theta(A).$$

We first emphasize the similarities with the framework presented in Chapter 3. The structure of actions is the very same, although the semantics of the query is different. Once more, Skolem terms in action effects makes the domain of the ABoxes obtained by executing actions continuously changing and in general unbounded in size and no persistence/frame assumption (except for equality, see later) is made. In principle at every move we substitute the whole old state, i.e., ABox, with a new one. Instead, differently to Chapter 3, we do have a persistence assumption on equalities, i.e., we implicitly copy all equalities holding in the current state to the new one. This implies that, as the system evolves, we acquire new information on equalities between terms, but never lose information on equalities already acquired.

Process. As we have seen in the previous artifact-centric systems, the process component can be seen as a possibly nondeterministic program that uses the data component to store its (intermediate and final) computation results, and the actions in as atomic instructions. Such a connotation also applies here. To specify such a process component, we adopt a rule-based specification.

Specifically, a *process* is a finite set Π of condition/action rules. A *condition/action rule* $\pi \in \Pi$ is an expression of the form

$$Q \mapsto \gamma$$

where γ is an action in Γ and Q is an ECQ, whose free variables are exactly the parameters of γ. The rule expresses that, for each tuple θ for which condition Q holds, the action γ with actual parameters θ *can* be executed. Processes do not force the execution of actions but constrain them: the user of the process will be able to choose any action that the rules forming the process allow. Moreover, our processes inherit entirely their states from the KAB knowledge component (TBox and ABox), see e.g., [44].

We remark again that we adopt a basic rule-based specification here because, in spite of its simplicity, a rule-based specification is able to expose all the difficulties of our setting. Other choices are also possible, in particular, the process could maintain its own state besides the one of the KAB. As long as such an additional state is finite, or embeddable into the KAB itself, the results here would easily extend to such a case.

Example 6.2.1. *Let us consider a KAB* $\mathcal{K} = (T, A_0, \Gamma, \Pi)$ *describing a superheroes comics world, where we have cities in which characters live. Figure 6.1 shows the TBox* T *and its rendering as a UML Class Diagram (see [21, 29] for the the relationship between UML Class Diagrams and Description Logics in general and DL-Lite in particular). Characters can be superheroes or (super)villains, who fight each other. As in the most classic plot, superheroes help the endeavors of law enforcement fighting villains threatening the city they live in. In fact, when a villain reveals himself for perpetrating his nefarious purposes against the city's peace, he consequently becomes a declared enemy of all superheroes living in that city. Each character lives in one city at the time. A common trait of almost all superheroes is a secret identity: a superhero is said to be the alter ego of some character, which is his identity in common life. Hence, the ABox assertion* $\mathsf{alterEgo}(s, p)$ *means that the superhero* s *is the alter ego of character* p*. Villains always try to unmask*

$$
\begin{aligned}
\mathsf{Villan} &\sqsubseteq \mathsf{Character}\\
\exists \mathsf{livesIn} &\sqsubseteq \mathsf{Character}\\
\exists \mathsf{livesIn}^- &\sqsubseteq \mathsf{City}\\
\mathsf{Character} &\sqsubseteq \exists \mathsf{livesIn}\\
(\mathsf{funct}\ \mathsf{livesIn})&\\
\exists \mathsf{enemy} &\sqsubseteq \mathsf{Villan}\\
\exists \mathsf{enemy}^- &\sqsubseteq \mathsf{Superhero}\\
\exists \mathsf{defeated} &\sqsubseteq \mathsf{Villan}\\
\exists \mathsf{defeated}^- &\sqsubseteq \mathsf{Superhero}\\
\mathsf{defeated} &\sqsubseteq \mathsf{enemy}\\
\exists \mathsf{alterEgo} &\sqsubseteq \mathsf{Superhero}\\
\exists \mathsf{alterEgo}^- &\sqsubseteq \mathsf{Character}\\
(\mathsf{funct}\ \mathsf{alterEgo})&
\end{aligned}
$$

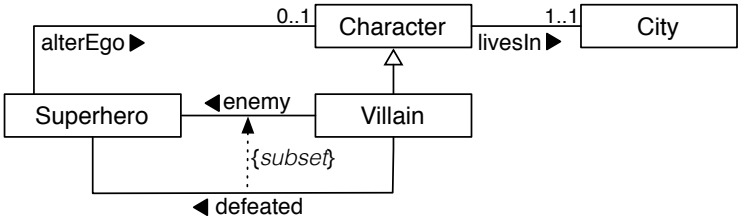


Figure 6.1. KAB's TBox for Example 6.2.1

superheroes, i.e., find their secret identity, in order to exploit such a knowledge to defeat them. Notice the subtle difference here: we use an $\mathsf{alterEgo}(s,p)$ *assertion to model the fact that* s *is the alter ego of* p*, whereas only by asserting* $s = p$ *we can capture the knowledge that* s *and* p *actually semantically denote the same individual.* Γ *may include actions like the following ones:*

$$\mathsf{BecomeSH}(p,c): \quad \{ \quad [\mathsf{Character}(p) \wedge \exists v.\mathsf{Villain}(v) \wedge \mathsf{livesIn}(v,c)] \rightsquigarrow \{\mathsf{Superhero}(\mathsf{sh}(p)), \mathsf{alterEgo}(\mathsf{sh}(p),p)\}, \; \mathit{CopyAll} \quad \}$$

states that if there exists at least one villain living in the city c*, a new superhero* $\mathsf{sh}(p)$ *can be created, with the purpose of protecting* c*. Such a superhero has* p *as alter ego. CopyAll is a shortcut for explicitly copying all concept and role assertions to the new state (equality assertions are always implicitly copied).*

$$\mathsf{Unmask}(s,p): \quad \{ \quad [\mathsf{alterEgo}(p,s)] \rightsquigarrow \{s = p\}, \; \mathit{CopyAll} \quad \}$$

states that superhero s*, who is the alter ego of* p*, gets unmasked by asserting the equality between* s *and* p *(it is now known that* $s = p$*).*

$$\mathsf{Fight}(v,s): \quad \{ \quad \exists p.[\mathsf{Villain}(v) \wedge \mathsf{Character}(p) \wedge \mathsf{alterEgo}(s,p)] \wedge [s = p] \rightsquigarrow \{\mathsf{defeated}(v,s)\}, \; \mathit{CopyAll} \quad \}$$

states that when villain v *fights superhero* s*, he defeats* s *if* s *has been unmasked, i.e., it is known that* s *is equal to his alter ego.*

$$\mathsf{Challenge}(v,s): \quad \{ \quad [\mathsf{Villain}(v) \wedge \mathsf{Superhero}(s) \wedge \exists p.\mathsf{alterEgo}(s,p) \wedge \mathsf{livesIn}(p,sc)] \wedge \neg[\mathsf{defeated}(v,s)] \rightsquigarrow \{\mathsf{livesIn}(v,sc), \mathsf{enemy}(v,s)\}, \; \mathit{CopyAll} \quad \}$$

states that when villain v *challenges superhero* s *and has not defeated him, next he lives in the same city as* s *and is an enemy of* s*.*

$$\mathsf{ThreatenCity}(v,c): \quad \{ \quad [\mathsf{Villain}(v) \wedge \mathsf{Superhero}(s) \wedge \exists p.\mathsf{alterEgo}(s,p) \wedge \mathsf{livesIn}(p,c)] \rightsquigarrow \{\mathsf{enemy}(v,s) \wedge \mathsf{livesIn}(v,c)\}, \; \mathit{CopyAll} \quad \}$$

states that when villain v threatens city c, then he becomes an enemy of all and only superheroes that live in c.

A process Π might include the following rules:

$$\begin{aligned}
[\mathsf{Character}(p)] \wedge \neg[\exists s.\mathsf{Superhero}(s) \wedge \mathsf{livesIn}(s,c)] &\mapsto \mathsf{BecomeSH}(p,c),\\
[\mathsf{Superhero}(s) \wedge \mathsf{Character}(c)] &\mapsto \mathsf{Unmask}(s,c),\\
[\mathsf{enemy}(v,s)] \wedge \neg[\exists v'.\mathsf{defeated}(v',s)] &\mapsto \mathsf{Fight}(v,s),\\
[\mathsf{Villain}(v) \wedge \mathsf{Superhero}(s)] &\mapsto \mathsf{Challenge}(v,s),\\
[\mathsf{Villain}(v) \wedge \mathsf{City}(c)] \wedge \neg\exists v'([\mathsf{Villain}(v') \wedge \mathsf{livesIn}(v',c)] \wedge \neg[v=v']) &\mapsto \mathsf{ThreatenCity}(v,c)
\end{aligned}$$

For instance, the first rule states that a character can become a superhero if the city does not already have one. While the last one states that a villan can threaten a city, if the city does not have another villan that is (known to be) distinct from him/her.

Notice that during the execution, reasoning on the KB is performed. For instance, consider an initial ABox:

$$\begin{aligned}
A_0 = \{\ &\mathsf{Superhero}(\mathsf{batman}),\ \mathsf{Villain}(\mathsf{joker}),\ \mathsf{alterEgo}(\mathsf{batman},\mathsf{bruce}),\\
&\mathsf{livesIn}(\mathsf{bruce},\mathsf{gotham}),\ \mathsf{livesIn}(\mathsf{batman},\mathsf{gotham}),\ \mathsf{livesIn}(\mathsf{joker},\mathsf{city1})\ \}
\end{aligned}$$

In this state, bruce *and* batman *live in the same city, and* batman *is the alterego of* bruce*, but it is not known whether they denote the same individual. Executing* $\mathsf{Challenge}(\mathsf{joker},\mathsf{batman})$ *in A_0, which is indeed allowed by the process Π, generates a new ABox with added assertions* $\mathsf{enemy}(\mathsf{joker},\mathsf{batman})$*,* $\mathsf{livesIn}(\mathsf{joker},\mathsf{gotham})$*, and* $\mathsf{gotham} = \mathsf{city1}$ *is implied by the functionality on* livesIn*.* □

6.3 KAB execution

As customary, the semantics of KABs is given in terms of possibly infinite transition systems that represent the possible evolutions of the KAB over time, as actions are executed according to the process. Notice that such transition systems must be equipped with semantically rich states, since a full KB is associated to them. Formally we define the kind of transition system we need as follows: A *transition system* is a tuple of the form $(\mathbb{U}, \mathcal{T}, \Sigma, s_0, \mathit{abox}, \Rightarrow)$, where:

- $\mathbb{U}$ is a countably infinite set of terms denoting individuals, called *universe*;
- $\mathcal{T}$ is a TBox;
- Σ is a set of states;
- $s_0 \in \Sigma$ is the initial state;
- *abox* is a function that, given a state $s \in \Sigma$, returns an ABox associated to s, which has as individuals terms of $\mathbb{U}$, and which conforms to $\mathcal{T}$;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

For convenience, we introduce the active domain of the whole transition system, defined as $\textsc{adom}() = \bigcup_{s\in\Sigma} \textsc{adom}(\mathit{abox}(s))$. Also we introduce the (predicate) alphabet $\textsc{alph}()$ of as the set of concepts and roles occurring in $\mathcal{T}$ or in the codomain of *abox*.

The KAB generates a transition system of this form during its execution. Formally, given a KAB, $\mathcal{K} = (\mathcal{T}, A_0, \Gamma, \Pi)$, we define its *(generated) transition system*, $\mathcal{K} = (\mathbb{U}, \mathcal{T}, \Sigma, s_0, \mathit{abox}, \Rightarrow)$ as follows:

- $\mathbb{U}$ is formed by all constants and all Skolem terms inductively formed starting from $\textsc{adom}(A_0)$ by applying the Skolem functions occurring in the actions in Γ;
- $\mathcal{T}$ is the TBox of the KAB;
- *abox* is the identity function (i.e., each state is simply an ABox);
- $s_0 = A_0$ is the initial state;
- Σ and $\Rightarrow$ are defined by mutual induction as the smallest sets satisfying the following property: if $s \in \Sigma$, then for each rule $Q \mapsto \gamma$, evaluate Q, and for each tuple θ returned, if $do(T, abox(s), \gamma\theta)$ is consistent w.r.t. $\mathcal{T}$, then $s \Rightarrow s'$, where $s' = do(T, abox(s), \gamma\theta)$.

Notice that the alphabet $\textsc{alph}(\mathcal{K})$ of $\mathcal{K}$ is simply formed by the set $\textsc{alph}(\mathcal{K})$ of concepts and roles that occur in $\mathcal{K}$.

The KAB transition system $\mathcal{K}$ is an infinite tree with infinitely many different ABoxes in its nodes, in general. In fact, to get a transition system that is infinite, it is enough to perform indefinitely a simple action that adds new terms at each step, e.g., an action of the form

$$\gamma() : \{\ [C(x)] \rightsquigarrow \{C(f(x))\},\quad \mathit{CopyAll}\ \}.$$

Hence the classical results on model checking [43], which are developed for finite transition systems, cannot be applied directly for verifying KABs.

6.4 Verification formalism

A first-order variant of μ-calculus is used to specify dynamic properties over KABs, where we allow local properties to be expressed as ECQs, and at the same time we allow for arbitrary first-order quantification across states. Given the nature of ECQs used for formulating local properties, first-order quantification ranges over terms denoting individuals.

Formally, we introduce the logic $\mu\mathcal{L}_A$ defined as follows:

$$\Phi \quad \longrightarrow \quad Q \ \mid\ \neg\Phi \ \mid\ \Phi_1 \wedge \Phi_2 \ \mid\ \exists x.\Phi \ \mid\ \langle - \rangle \Phi \ \mid\ Z \ \mid\ \mu Z.\Phi$$

where Q is a possibly open ECQ, and Z is a second order predicate variable (of arity 0). We make use of the following abbreviations: $\forall x.\Phi = \neg(\exists x.\neg\Phi)$, $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$, $[\Phi] = \neg\langle - \rangle \neg\Phi$, and $\nu Z.\Phi = \neg\mu Z.\neg\Phi[Z/\neg Z]$. The formulae $\mu Z.\Phi$ and $\nu Z.\Phi$ respectively denote the least and greatest fixpoint of the formula Φ (seen as the predicate transformer $\lambda Z.\Phi$). As usual in μ-calculus, formulae of the form $\mu Z.\Phi$ (and $\nu Z.\Phi$) must obey to the *syntactic monotonicity* of Φ w.r.t. Z, which states that every occurrence of the variable Z in Φ must be within the scope of an even number of negation symbols. This ensures that the least fixpoint $\mu Z.\Phi$ (as well as the greatest fixpoint $\nu Z.\Phi$) always exists.

The semantics of $\mu\mathcal{L}_A$ formulae is defined over possibly infinite transition systems of the form $\langle \mathbb{U}, \mathcal{T}, \Sigma, s_0, abox, \Rightarrow \rangle$ seen above. Since $\mu\mathcal{L}_A$ also contains formulae with both individual and predicate free variables, given a transition system , we introduce an *individual variable valuation* v, i.e., a mapping from individual variables

x to $\mathbb{U}$, and a *predicate variable valuation* V, i.e., a mapping from the predicate variables Z to subsets of Σ. With these three notions in place, we assign meaning to formulae by associating to Υ, v, and V an *extension function* $(\cdot)^{\Upsilon}_{v,V}$, which maps formulae to subsets of Σ. Formally, the extension function $(\cdot)^{\Upsilon}_{v,V}$ is defined inductively as follows:

$$\begin{array}{rcl}
(Q)^{\Upsilon}_{v,V} & = & \{s \in \Sigma \mid \textsc{ans}(Qv, T, abox(s)) = true\}, \\
(\neg\Phi)^{\Upsilon}_{v,V} & = & \Sigma \setminus (\Phi)^{\Upsilon}_{v,V}, \\
(\Phi_1 \wedge \Phi_2)^{\Upsilon}_{v,V} & = & (\Phi_1)^{\Upsilon}_{v,V} \cap (\Phi_2)^{\Upsilon}_{v,V}, \\
(\exists x.\Phi)^{\Upsilon}_{v,V} & = & \{s \in \Sigma \mid \exists t.t \in \textsc{adom}(abox(s)) \text{ and } s \in (\Phi)^{\Upsilon}_{v[x/t],V}\}, \\
(\langle - \rangle \Phi)^{\Upsilon}_{v,V} & = & \{s \in \Sigma \mid \exists s'.s \Rightarrow s' \text{ and } s' \in (\Phi)^{\Upsilon}_{v,V}\}, \\
(Z)^{\Upsilon}_{v,V} & = & V(Z), \\
(\mu Z.\Phi)^{\Upsilon}_{v,V} & = & \bigcap\{\mathcal{E} \subseteq \Sigma \mid (\Phi)^{\Upsilon}_{v,V[Z/\mathcal{E}]} \subseteq \mathcal{E}\}.
\end{array}$$

Here Qv stands for the (boolean) ECQ obtained from Q by substituting its free variables according to v. When Φ is a closed formula, $(\Phi)^{\Upsilon}_{v,V}$ does not depend on v or V, and we denote the extension of Φ simply by $(\Phi)^{\mathfrak{u}}$. A closed formula Φ holds in a state $s \in \Sigma$ if $s \in (\Phi)^{\mathfrak{u}}$. In this case, we write $\Upsilon, s \models \Phi$. A closed formula Φ holds in Υ, denoted by $\Upsilon \models \Phi$, if $\Upsilon, s_0 \models \Phi$. We call *model checking* the problem of verifying whether $\Upsilon \models \Phi$ holds.

We refer to Section 2.3 for the intuitive meaning of the extension function $(\cdot)^{\Upsilon}_{v,V}$. Notice that here, differently from languages in Chapter 2 and 3, we do have (a restricted form of) quantification across state, meaning that fixpoint operators may occur in the scope of quantifiers. Such a quantification is "restricted" in that it ranges over element of the active domain of the current state.

The next example shows some simple temporal properties that can be expressed in $\mu\mathcal{L}_A$.

Example 6.4.1. *Considering the KAB of Example 6.2.1, we can easily express temporal properties as the following ones.*

- *From now on all current superheroes that live in Gotham will live in Gotham forever (a form of safety):*

$$\forall x.[\mathsf{Superhero}(x) \wedge \mathsf{livesIn}(x, \mathsf{gotham})] \rightarrow \nu Z.([\mathsf{livesIn}(x, \mathsf{gotham})] \wedge [Z])$$

- *Eventually all current superheroes will be unmasked (a form of liveness):*

$$\forall x.[\mathsf{Superhero}(x)] \rightarrow \mu Z.([\mathsf{alterEgo}(x, x)] \vee [Z])$$

- *There exists a possible future situation where all current superheroes will be unmasked (another form of liveness):*

$$\forall x.[\mathsf{Superhero}(x)] \rightarrow \mu Z.([\mathsf{alterEgo}(x, x)] \vee \langle - \rangle Z)$$

- *Along every future, it is always true, for every superhero, that there exists an evolution that eventually leads to unmask him (a form of liveness that holds in every moment):*

$$\nu Y.(\forall x.[\mathsf{Superhero}(x)] \rightarrow \mu Z.([\mathsf{alterEgo}(x, x)] \vee \langle - \rangle Z)) \wedge [Y]$$

■

Consider two transition systems sharing the same universe and the same predicate alphabet. We say that they are *behaviourally equivalent* if they satisfy exactly the same $\mu\mathcal{L}_A$ formulas. To formally capture such an equivalence, we make use of the notion of *bisimulation* [108], suitably extended to deal with query answering over KBs.

Given two transition systems $\mathtt{1} = \langle \mathbb{U}, \mathcal{T}, \Sigma_1, s_{01}, abox_1, \Rightarrow_1 \rangle$ and $\mathtt{2} = \langle \mathbb{U}, \mathcal{T}, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$, sharing the same universe $\mathbb{U}$, the same TBox T, and such that $\textsc{alph}(\mathtt{1}) = \textsc{alph}(\mathtt{2}) = \Lambda$, a *bisimulation* between $\mathtt{1}$ and $\mathtt{2}$ is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $(s_1, s_2) \in \mathcal{B}$ implies that:

1. $abox(s_1) \equiv_{T,\Lambda} abox(s_2)$;
2. if $s_1 \Rightarrow_1 s_1'$, then there exists s_2' such that $s_2 \Rightarrow_2 s_2'$ and $(s_1', s_2') \in \mathcal{B}$;
3. if $s_2 \Rightarrow_2 s_2'$, then there exists s_1' such that $s_1 \Rightarrow_1 s_1'$ and $(s_1', s_2') \in \mathcal{B}$.

We say that two states s_1 and s_2 are *bisimilar*, if there exists a bisimulation $\mathcal{B}$ such that $(s_1, s_2) \in \mathcal{B}$. Two transition systems $\mathtt{1}$ with initial state s_{01} and $\mathtt{2}$ with initial state s_{02} are *bisimilar* if $(s_{01}, s_{02}) \in \mathcal{B}$. The following theorem states that the formula evaluation in $\mu\mathcal{L}_A$ is indeed invariant w.r.t. bisimulation, so we can equivalently check any bisimilar transition systems.

Theorem 6.3. *Let $\mathtt{1}$ and $\mathtt{2}$ be two transition systems that share the same universe, the same TBox, and the same predicate alphabet, and that are bisimilar. Then, for two states s_1 of $\mathtt{1}$ and s_2 of $\mathtt{2}$ (including the initial ones) that are bisimilar, and for all closed $\mu\mathcal{L}_A$ formulas Φ, we have that*

$$s_1 \in (\Phi)^{\mathtt{1}} \quad \textit{iff} \quad s_2 \in (\Phi)^{\mathtt{2}}.$$

Proof. The proof is analogous to the standard proof of bisimulation invariance of μ-calculus [125], though taking into account our bisimulation, which guarantees that ECQs are evaluated identically over bisimilar states. Notice that the assumption that the two transition systems share the same universe and the same predicate alphabet makes it easy to compare the answers to queries. □

Making use of such a notion of bisimulation, we can, for example, redefine the transition system generated by a KAB $\mathcal{K} = (\mathcal{T}, A_0, \Gamma, \Pi)$ while maintaining bisimilarity, by modifying the definition of $\mathcal{K} = \langle \mathbb{U}, \mathcal{T}, \Sigma, s_0, abox, \Rightarrow \rangle$ given in Section 6.3 as follows.

(i) We modify $do()$ so that no Skolem term t' is introduced in the generated ABox A' if in the current ABox (note that all terms that are present in the current ABox are preserved in the new ABox, together with equalities between terms) A there is already a term t such that $(T, A) \models t = t'$.

(ii) If the ABox $A' = do(T, abox(s), \gamma\theta)$ obtained from the current state s is logically equivalent to the ABox $abox(s'')$, for some already generate state s'', we do not generate a new state, but simply add $s \Rightarrow s''$ to $\mathcal{K}$.

6.5 Decidability

It is immediate to see that verification of KABs is undecidable in general. Indeed, it is easy to represent Turing machines using a KAB. In fact, we can do so using only a fragment of the capabilities of KABs, as shown in the next lemma.

Lemma 6.1. *Checking formulae of the form $\mu Z.(N(a) \vee \langle - \rangle Z)$, where N is an atomic concept and a is an individual occurring in A_0, is undecidable already for a KAB $\mathcal{K} = (\mathcal{T}, A_0, \Gamma, \Pi)$ where:*

- *$\mathcal{T}$ is the empty TBox,*
- *the actions in Γ make no use of negation nor equality,*
- *Π is the trivial process that poses no restriction on executability of actions.*

Proof. Given a Turing machine $\mathcal{M} = \langle Q, \Sigma, q_0, \delta, q_f, \rangle$, we show how to construct a corresponding KAB $\mathcal{K}_\mathcal{M} = (\emptyset, A_0, \Gamma, \Pi)$ that mimics the behavior of $\mathcal{M}$. Specifically, we encode the halting problem for $\mathcal{M}$ as a verification problem over $\mathcal{K}_\mathcal{M}$. Roughly speaking, $\mathcal{K}_\mathcal{M}$ maintains the tape and state information in the (current) ABox, and encodes the transitions of $\mathcal{M}$ as actions. Our construction makes use of a tape that initially contains a unique cell, represented by the constant $\mathsf{0}$, and is extended on-the-fly as needed: cells to the right of $\mathsf{0}$ are represented by Skolem terms of the form $\mathsf{n}(\mathsf{n}(\cdots(\mathsf{0})\cdots))$, while cells to the left of $\mathsf{0}$ are represented by Skolem terms of the form $\mathsf{p}(\mathsf{p}(\cdots(\mathsf{0})\cdots))$. Then, we make use of one constant a_q for each state $q \in Q$, of one constant a_v for each tape symbol value $v \in \Sigma$, of a special constant $\mathtt{\#}$, and of the following concepts and roles:

- $\mathsf{cell}(c, h)$ models a cell of the tape, where c is a cell identifier, and h corresponds to the current state of $\mathcal{M}$, if the head of $\mathcal{M}$ currently points to c, or to $\mathtt{\#}$ if the head does not currently point to c;
- $\mathsf{next}(c_l, c_r)$ models the relative position of cells, stating that c_r is the cell immediately following c_l;
- $\mathsf{value}(c, v)$ models that cell c currently contains value v, with $v \in \Sigma$;
- $\mathsf{First}(c)$ and $\mathsf{Last}(c)$ respectively denote the current first cell and last cell of the tape.
- $\mathsf{Stop}(c)$ is used to detect when $\mathcal{M}$ halts.

The initial state of $\mathcal{K}_\mathcal{M}$ contains a unique cell and is defined as

$$A_0 = \{\ \mathsf{cell}(\mathsf{0}, \mathsf{a}_{q_0}),\ \mathsf{value}(\mathsf{0}, \mathsf{a}),\ \mathsf{First}(\mathsf{0}),\ \mathsf{Last}(\mathsf{0})\ \}$$

As for the action component, Γ contains an action with no parameters for each transition in δ, while the process Π poses no restriction on executability of actions, i.e., it contains a rule $\mathtt{true} \mapsto \gamma()$ for each such action γ.

We now provide the specification of actions, detailing the case of a right shift transition $\delta(q, v, q', v', R)$. The corresponding action specification consists of the set of effects shown in Figure 6.2. The first effect maintains the first position of the tape unaltered. The second and third effects deal with the cell values. They remain the same except for the current cell, which is updated according to the transition. The next three effects deal with the right shift and the Turing Machine state. If the current cell has a next cell and therefore is not the last one, then the head is moved to the next cell and the state change of $\mathcal{M}$ is recorded there. In this case the last cell remains the same. If instead the current cell is the last one, before moving the head the tape must be properly extended. The Skolem function $\mathsf{n}/1$ is used to create the identifier of this new successor cell, starting from the identifier of the current one. Furthermore, since the transition corresponds to a right shift

$$
\begin{array}{rcl}
[\mathsf{First}(c)] & \rightsquigarrow & \{\mathsf{First}(c)\} \\
[\mathsf{cell}(c,\#) \wedge \mathsf{value}(c,x)] & \rightsquigarrow & \{\mathsf{value}(c,x)\} \\
[\mathsf{cell}(c,\mathsf{a}_q) \wedge \mathsf{value}(c,\mathsf{a}_v)] & \rightsquigarrow & \{\mathsf{value}(c,\mathsf{a}_{v'})\} \\
[\mathsf{cell}(c,\mathsf{a}_q) \wedge \mathsf{value}(c,\mathsf{a}_v) \wedge \mathsf{next}(c,c_r)] & \rightsquigarrow & \{\mathsf{cell}(c_r,\mathsf{a}_{q'})\} \\
[\mathsf{cell}(c,\mathsf{a}_q) \wedge \mathsf{value}(c,\mathsf{a}_v) \wedge \mathsf{Last}(c)] & \rightsquigarrow & \{\mathsf{cell}(\mathsf{n}(c),\mathsf{a}_{q'}), \mathsf{next}(c,\mathsf{n}(c)), \mathsf{Last}(\mathsf{n}(c))\} \\
[\mathsf{cell}(c,\#) \wedge \mathsf{Last}(c)] & \rightsquigarrow & \{\mathsf{Last}(c)\} \\
[\mathsf{cell}(c,\#) \wedge \mathsf{First}(c)] & \rightsquigarrow & \{\mathsf{cell}(c,\#)\} \\
[\mathsf{cell}(c,\#) \wedge \mathsf{next}(c,c_r)] & \rightsquigarrow & \{\mathsf{cell}(c_r,\#)\} \\
[\mathsf{cell}(c,\mathsf{a}_{q_f})] & \rightsquigarrow & \{\mathsf{Stop}(0)\}
\end{array}
$$

Figure 6.2. Effects of the action used to encode a transition $\delta(q, v, q', v', R)$ of a Turing Machine

of one cell, the first cell and all the cells immediately following a cell marked # will be marked # in the next state. Finally, the last effect is used to identify the case in which $\mathcal{M}$ has reached a final state. This is marked by inserting into the new state the special assertion $\mathsf{Stop}(0)$.

The construction for a left shift transition is done symmetrically, using the Skolem function $\mathsf{p}/1$ to create a new predecessor cell. By construction, $\mathcal{K}_{\mathcal{M}}$ satisfies the conditions of the theorem. Observe that, in the transition system $\mathcal{K}_{\mathcal{M}}$ generated by $\mathcal{K}_{\mathcal{M}}$, every action corresponding to every transition of $\mathcal{M}$ can be executed in each ABox/state s of $\mathcal{K}_{\mathcal{M}}$, and since $\mathcal{T}$ is empty, it will actually generate a successor state of s. However, in each state, only the (unique) action that corresponds to the actually executed transition of $\mathcal{M}$ will generate a successor state containing an ABox assertion of the form $\mathsf{cell}(c, \mathsf{a}_q)$, for some state q of $\mathcal{M}$. Therefore, only those ABoxes/states properly corresponding to configurations of $\mathcal{M}$ could eventually lead to an ABox/state in $\mathcal{K}_{\mathcal{M}}$ where $\mathsf{Stop}(0)$ holds. And the latter will happen if and only if $\mathcal{M}$ halts. More precisely, once can show by induction on the lenght respectively of a halting computation of $\mathcal{M}$ and of the shortest path from the initial state of $\mathcal{K}_{\mathcal{M}}$ to a state where $\mathsf{Stop}(0)$ holds, that $\mathcal{M}$ halts if and only if $\mathcal{K}_{\mathcal{M}} \models \mu Z.([\mathsf{Stop}(0)] \vee \langle - \rangle Z)$, which concludes the proof. □

From the previous lemma, which shows undecidability already in a special case, we immediately obtain the following result.

Theorem 6.4. *Verification of $\mu\mathcal{L}_A$ formulae over KABs is undecidable.*

We observe that Lemma 6.1 uses a KB that is constituted only by an ABox containing concept and role assertions, and makes use only of conjunctive queries in defining actions effects. Moreover, the formula that we check makes no use of quantification at all, and can simply be seen as a propositional CTL formula of the form EFp, expressing that proposition p eventually holds along one path.

In spite of Theorem 6.4, next we introduce a notable class of KABs for which verification of arbitrary $\mu\mathcal{L}_A$ properties is decidable. To do so, we rely on a syntactic restriction that resembles the notion of *weak acyclicity* in data exchange [65], and that guarantees boundedness of ABoxes generated by the execution of the KAB, and in turn decidability of verification. We use, once more, the original definition of

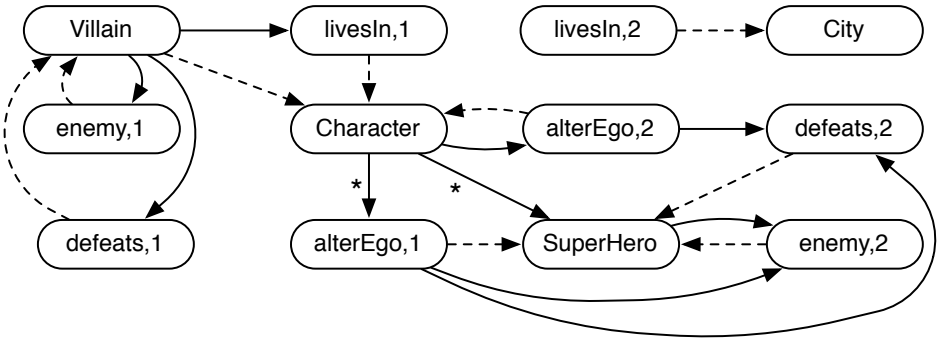


Figure 6.3. Weakly acyclic dependency graph for Example 6.2.1.

weak acyclicity. However, our results can be applied also to other variants of weak acyclicity (cf. Section 2.4)

Now we are ready to introduce the notion of weak acyclicity in our context. The careful reader will notice that the same concepts of Section 2.4, but now tailored on KAB, are also used here.

The edge-labeled directed *dependency graph* of a KAB $\mathcal{K} = (T, A_0, \Gamma, \Pi)$, defined as follows. *Nodes*, called *positions*, are obtained from the TBox T: there is a node for every concept name N in T, and two nodes for every role name P in T, corresponding to the domain and to the range of P. *Edges* are drawn by considering every effect specification $[q^+] \wedge Q^- \rightsquigarrow A'$ of each action contained in Γ, tracing how values are copied or contribute to generate new values as the system progresses. In particular, let p be a position corresponding to a concept/role component in the rewriting $rew(q^+)$ of q^+ with variable x. For every position p' in A' with the same variable x, we include a *normal edge* $p \to p'$. For every position p'' in A' with a Skolem term $f(\vec{t})$ such that $x \in \vec{t}$, we include a *special edge* $p \xrightarrow{*} p''$. We say that $\mathcal{K}$ is *weakly-acyclic* if its dependency graph has no cycle going through a special edge.

Example 6.5.1. *The KAB of Example 6.2.1 is weakly acyclic. Its dependency graph, shown in Figure 6.3, does not contain any cycle going through special edges. For readability, self-loops are not shown in the Figure (but are present for all nodes), and dashed edges are used to compactly represent the contributions given by the rewriting of the queries. E.g., the dashed edge form* Villain *to* Character *denotes that for every outgoing edge from* Character*, there exists an outgoing edge from* Villain *with the same type and target. Hence, w.r.t. weak acyclicity dashed edges can be simply replaced by normal edges.* ■

We are now ready to state the main result of this work, which we are going to prove in the remainder of this section.

Theorem 6.5. *Verification of $\mu\mathcal{L}_A$ properties for a weakly acyclic KAB is decidable in* ExpTime *in the size of the KAB.*

We observe that the restriction imposed by weak acyclicity (or variants) is not too severe, and in many real cases KABs are indeed weakly acyclic or can be transformed into weakly acyclic ones at cost of redesign. Indeed, weakly acyclic KABs cannot indefinitely generate new values from the old ones, which then depend on a

chain of unboundedly many previous values. In other words, current values depend only on a bounded number of old values. While unbounded systems exist in theory, e.g., Turing machines, higher level processes, as those in business process management or service-oriented modeling, typically require such a boundedness in practice. How to systematically transform systems into weakly acyclic ones remains an open issue.

In the remainder of this section we present the proof of Theorem 6.5. We do so in several steps:

1. **Normalized KAB.** First we introduce a normalized form $\hat{\mathcal{K}}$ of the KAB $\mathcal{K}$, which isolates the contribution of equalities and of the TBox in actions effects of the KAB. An important point is that normalizing the KAB preserves weak acyclicity.

2. **Normalized** $do()$**.** Then, we introduce a normalized version $\textsc{do}_{\text{SMP}}()$ of $do()$, which avoids to consider equalities in generating the bulk set of tuples to be used in the effects to generate the next ABox. The transition system $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$ generated through this normalized version $\textsc{do}_{\text{SMP}}()$ of $do()$ by the normalized KAB $\hat{\mathcal{K}}$ is bisimilar to the transition system $\mathcal{K}$ generated through $do()$ by $\mathcal{K}$. Hence the two transition systems satisfy the same $\mu\mathcal{L}_A$ formulae.

3. **Positive dominant.** The next step is to introduce what we call the *positive dominant* $\mathcal{K}^{++}$ of the normalized KAB $\hat{\mathcal{K}}$. This is obtained from $\mathcal{K}^{++}$ essentially by dropping equalities, negations, and TBox. However $\mathcal{K}^{++}$ contains enough information in the positive part so that, when we drop all of these features, the active domain of the transition system $\mathcal{K}^{++}$ generated by $\mathcal{K}^{++}$ "overestimates" the active domain of the transition system $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$ generated by the normalized KAB $\hat{\mathcal{K}}$. Moreover if the normalized (and hence the original) KAB is weakly acyclic, then so is its positive dominant. Finally if the positive dominant is weakly acyclic then the size of the active domain of its transition system $\mathcal{K}^{++}$ is polynomially bounded by the size of its initial ABox, and hence so is the size of the active domain of $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$. This implies that the size of $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$ is finite and at most exponential in the size of its initial ABox.

4. **Putting it all together.** Tying these results together, we get the claim.

In the following, we detail each of these steps.

6.5.1 Normalized KAB

Given a KAB $\mathcal{K} = (\mathcal{T}, A_0, \Gamma, \Pi)$, we build a KAB $\hat{\mathcal{K}} = (\mathcal{T}, \hat{A}_0, \hat{\Gamma}, \Pi)$, called the *normalized form of* $\mathcal{K}$, by applying a sequence of transformations that preserve the semantics of $\mathcal{K}$, while producing a KAB of a format that is easier to study.

1. In $\hat{\mathcal{K}}$, all individuals appearing in equality assertions in an ABox also occur in special concept assertions of the form $Dummy(t)$ where the concept $Dummy$ is unrelated to the other concepts and roles in the KAB. We do so by:
   - adding concept assertions $Dummy(t)$ for each t in an equality assertion in A_0 that does not appear elsewhere;

- adding to the right-hand part of each action effect e_i a concept assertions $Dummy(t)$ for each t in an equality assertion in the right-hand part of e_i;
- adding to each action an effect specification of the form

$$[Dummy(x)] \rightsquigarrow \{Dummy(x)\}$$

Notice that, as the result of this transformation, we get ABoxes containing the additional concept *Dummy*, which however is never queried by actions effects and by the rules forming the process. The impact of the transformation is simply that now the $\text{ADOM}(A)$ of the ABoxes A in the KAB transition system can be readily identified as the set of terms occurring in concept and role assertion only (without looking at equality assertions).

2. We view each ABox A as partitioned into a part collecting concept and role assertions, denoted by $A^{\not\text{EQ}} = A \setminus \text{EQ}(\mathcal{T}, A)$, and a part, denoted as $\text{EQ}(\mathcal{T}, A)$, collecting only equality assertions, which, for convenience, we assume closed w.r.t. the TBox T. That is:

$$A = A^{\not\text{EQ}} \cup \text{EQ}(\mathcal{T}, A).$$

Given an ABox A, we denote by $\hat{A}$ the result of the two above transformations, which respectively add to A the extension of *Dummy* and closure of equalities.

3. We manipulate each effect specification

$$[q^+] \wedge Q^- \rightsquigarrow \hat{A}'$$

of actions appearing in Γ as follows:

3.1. We replace $[q^+] \wedge Q^-$ by $[rew(q^+)] \wedge rew(Q^-)$ [30], exploiting the results in [31, 5], which guarantee that, for every ECQ Q,

$$\text{ANS}(Q, T, A) = \text{ANS}(rew(Q), \emptyset, A).$$

3.2. We replace each effect specification $[rew(q^+)] \wedge rew(Q^-) \rightsquigarrow \hat{A}'$, resulting from Step 3.1, by a set of effect specifications $[q_i^+] \wedge rew(Q^-) \rightsquigarrow \hat{A}'$, one for each CQ q_i in the UCQ $rew(q^+)$.

3.3. For each effect specification $[q_i^+] \wedge rew(Q^-) \rightsquigarrow \hat{A}'$, we re-express q_i^+ so as to make equalities used to join terms explicit and so as to remove constants from q_i^+. Specifically, we replace the effect specification by

$$[q_i^{++}] \wedge q^{=} \wedge rew(Q^-) \rightsquigarrow \hat{A}',$$

where:

- q_i^{++} is the CQ without repeated variables obtained from q_i^+ by *(i)* replacing for each variable x occurring in q_i^+, the j-th occurrence of x except for the first one, by $x^{[j]}$; and *(ii)* replacing each constant c with a new variable x_c;

- $q^{=} = \bigwedge[x = x^{[j]}] \wedge \bigwedge[x_c = c]$ where *(i)* the first conjunction contains one equality $[x = x^{[j]}]$ for each variable x in q_i^+ and for each variable $x^{[j]}$ introduced in the step above, and *(ii)* the second conjunction contains one equality for each constant c in q_i^+.

To clarify the latter consider the following example:

Example 6.5.2. *Given a query*

$$[q_i^+] \doteq [N(x) \wedge P_1(x, y) \wedge P_2(c, x)],$$

Step 3.3 above replaces it by $[q_i^{++}] \wedge q^{=}$*, where*

$$q_i^{++} \doteq N(x) \wedge P_1(x^{[2]}, y) \wedge P_2(x_c, x^{[3]}), \qquad q^{=} \doteq [x = x^{[2]}] \wedge [x = x^{[3]}] \wedge [x_c = c]$$

■

As for the correctness of Step 3.3, it is immediate to notice that $[q_i^+]$ is equivalent to $[q_i^{++} \wedge \bigwedge(x = x^{[j]}) \wedge \bigwedge(x_c = c)]$. The equivalence between the latter and $[q_i^{++}] \wedge q^{=}$ is a consequence of the construction in [5], which shows that query entailment in the presence of equalities can be reduced to query evaluation by saturating equalities w.r.t. transitivity, reflexivity, symmetry, and functionality.

Given an action γ, we denote by $\hat{\gamma}$ the action normalized as above.

Since all transformations preserve logical equivalence (as long as we do not query *Dummy*), we have

Lemma 6.2. $do(T, A, \gamma\theta) \equiv_{T,\text{ALPH}(\mathcal{K})} do(T, \hat{A}, \hat{\gamma}\theta)$.

Also the normalization of a KAB preserves weak acyclicity, which is a crucial consideration for later results.

Lemma 6.3. *If $\mathcal{K}$ is weakly acyclic then also $\hat{\mathcal{K}}$ is weakly acyclic.*

Proof. Consider each effect specification $[q^+] \wedge Q^- \rightsquigarrow A'$ belonging to an action in $\mathcal{K}$. The contribution of this effect specification to the dependency graph $\mathcal{G}$ of $\mathcal{K}$ is limited to each CQ q_i in the UCQ $rew(q^+)$, and to the set of concept and role assertions of A'. We observe that each such q_i corresponds to a query q_i^{++} in $\hat{\mathcal{K}}$ in which each variable of q_i occurs exactly once. For every free variable x of q_i that also appears in $A'i$, and for every occurrence of x in q_i itself, an edge is included in $\mathcal{G}$. In the dependency graph $\hat{\mathcal{G}}$ of $\hat{\mathcal{K}}$, only one of such edges appears, corresponding to the single occurrence of the variable x in q_i^{++}.

Notice that *Dummy* in the alphabet of $\hat{\mathcal{K}}$ but not in the alphabet of $\mathcal{K}$ can be omitted from the dependency graph of $\hat{\mathcal{G}}$ since by definition of $\hat{\mathcal{K}}$, *Dummy* does not occur in the left-hand side of effects except for the trivial effect $[Dummy(x)] \rightsquigarrow \{Dummy(x)\}$. Therefore, $\hat{\mathcal{G}}$ is indeed a subgraph of $\mathcal{G}$, and hence weak acyclicity of $\mathcal{G}$ implies weak acyclicity of $\hat{\mathcal{G}}$. □

6.5.2 Normalized $do()$

Next we give a simplified version of $do()$, which we call $\text{DO}_{\text{SMP}}()$. We start by observing that can reformulate the definition of $do()$ given in Section 6.2. In particular, given an action $\hat{\gamma}$ with parameters substitution θ and an ABox $\hat{A}$, we have

$$do(\mathcal{T}, \hat{A}, \hat{\gamma}\theta) = \bigcup_{e \in \hat{\gamma}} \text{APPLY}(\mathcal{T}, \hat{A}, e, \theta),$$

where for an effect specification $e : [q^{++}] \wedge q^{=} \wedge Q^{-} \rightsquigarrow \hat{A}'$, we have

$$\text{APPLY}(\mathcal{T}, \hat{A}, e, \theta) = \bigcup_{\sigma \in \text{ANS}(q^{++}\theta, \emptyset, \hat{A}) \cap \text{ANS}((q^{=} \wedge Q^{-})\theta, \emptyset, \hat{A})} \hat{A}'\theta\sigma \cup \text{EQ}(\mathcal{T}, \hat{A}).$$

Instead, we define $\text{DO}_{\text{SMP}}()$ as:

$$\text{DO}_{\text{SMP}}(\mathcal{T}, \hat{A}, \hat{\gamma}\theta) = \bigcup_{e \in \hat{\gamma}} \text{APPLY}_{\text{SMP}}(\mathcal{T}, \hat{A}, e, \theta),$$

where, for an effect specification $e : [q^{++}] \wedge q^{=} \wedge Q^{-} \rightsquigarrow \hat{A}'$, we have

$$\text{APPLY}_{\text{SMP}}(\mathcal{T}, \hat{A}, e, \theta) = \bigcup_{\sigma \in \text{ANS}(q^{++}\theta, \emptyset, \hat{A}^{\not\exists Q}) \cap \text{ANS}((q^{=} \wedge Q^{-})\theta, \emptyset, \hat{A})} \hat{A}'\theta\sigma \cup \text{EQ}(\mathcal{T}, \hat{A}).$$

Notice that the only difference between $do()$ and $\text{DO}_{\text{SMP}}()$ is that in the latter we use only $\hat{A}^{\not\exists Q}$ instead of $\hat{A}$ to compute the answers to the CQs $q^{++}\theta$.

The following lemma shows that the application of $do()$ and $\text{DO}_{\text{SMP}}()$ gives rise to logically equivalent ABoxes.

Lemma 6.4. $do(\mathcal{T}, \hat{A}, \hat{\gamma}\theta) \equiv_{\mathcal{T}, \text{ALPH}(\mathcal{K})} \text{DO}_{\text{SMP}}(\mathcal{T}, \hat{A}, \hat{\gamma}\theta)$.

Proof. In order to prove the claim, it is enough to show that for each concept/role assertion $\alpha_2 \in \text{DO}_{\text{SMP}}(\mathcal{T}, \hat{A}, \hat{\gamma}\theta)$ whose concept/role name belongs to $\text{ALPH}(\mathcal{K})$, we have that $\mathcal{T}, do(\mathcal{T}, \hat{A}, \hat{\gamma}\theta) \models \alpha_2$, and for each concept/role assertion $\alpha_1 \in do(\mathcal{T}, \hat{A}, \hat{\gamma}\theta)$ whose concept/role name belongs to $\text{ALPH}(\mathcal{K})$, we have that $\mathcal{T}, \text{DO}_{\text{SMP}}(\mathcal{T}, \hat{A}, \hat{\gamma}\theta) \models \alpha_1$. We actually prove a slightly stronger result:

(1) for each ABox assertion $\alpha_2 \in \text{APPLY}_{\text{SMP}}(\mathcal{T}, \hat{A}, e, \theta)$, we have that $\mathcal{T}, \text{APPLY}(\mathcal{T}, \hat{A}, e, \theta) \models \alpha_2$;

(2) for each ABox assertion $\alpha_1 \in \text{APPLY}(\mathcal{T}, \hat{A}, e, \theta)$, we have that $\mathcal{T}, \text{APPLY}_{\text{SMP}}(\mathcal{T}, \hat{A}, e, \theta) \models \alpha_1$.

For *(1)*, by monotonicity of q^{++} and the fact that $\hat{A}^{\not\exists Q} \subseteq \hat{A}$, we have that

$$\bigcup_{\sigma \in (\text{ANS}(q^{++}\theta, \emptyset, \hat{A}^{\not\exists Q}) \cap \text{ANS}((q^{=} \wedge Q^{-})\theta, \emptyset, \hat{A}))} \hat{A}'\theta\sigma \quad \text{is contained in} \quad \bigcup_{\sigma \in (\text{ANS}(q^{++}\theta, \emptyset, \hat{A}) \cap \text{ANS}((q^{=} \wedge Q^{-})\theta, \emptyset, \hat{A}))} \hat{A}'\theta\sigma$$

hence the claim follows.

For *(2)*, consider an ABox assertion $\alpha \in \text{APPLY}(\mathcal{T}, \hat{A}, e, \theta)$. By definition of $\text{APPLY}()$, we know that there exists an effect $e : [q^{++}] \wedge Q^{-} \wedge q^{=} \rightsquigarrow \hat{A}'$ and an assignment σ to the free variables of q^{++} such that $\sigma \in (\text{ANS}(q^{++}\theta, \emptyset, \hat{A}) \cap \text{ANS}((q^{=} \wedge$

$Q^-)\theta, \emptyset, \hat{A}))$ and $\alpha \in \hat{A}'\theta\sigma$. Let $\{x_1, \ldots, x_n\}$ be all free variables in $q^{++}\theta$, and $\sigma = \{x_1 \to t_1, \ldots, x_n \to t_n, \}$. For each variable x_i, let $N(x_i)$ be the (unique) concept atom in $q^{++}\theta$ in which x_i occurs (similar considerations hold when x_i occurs in a role atom). Then, either $N(t_i) \in \hat{A}^{\not\in Q}$, or for some t'_i, $N(t'_i) \in \hat{A}^{\not\in Q}$ and $(t_i = t'_i) \in \text{EQ}(\mathcal{T}, \hat{A})$. In the former case, let t''_i denote t_i, while in the latter case let t''_i denote t'_i. Then, consider the substitution $\sigma' = \{x_1 \to t''_1, \ldots, x_n \to t''_n, \}$. By construction, we have that $\sigma' \in \text{ANS}(q^{++}\theta, \emptyset, \hat{A}^{\not\in Q})$, and since $\sigma \in \text{ANS}((q^{=} \wedge Q^-)\theta, \emptyset, \hat{A})$, and $(t''_i = t_i) \in \text{EQ}(\mathcal{T}, \hat{A})$ for each $i \in \{1, \ldots, n\}$, we also have that $\sigma' \in \text{ANS}(q^{++}\theta, \emptyset, \hat{A}^{\not\in Q}) \cap \text{ANS}((q^{=} \wedge Q^-)\theta, \emptyset, \hat{A})$. Since

- $\alpha \in \hat{A}'\theta\sigma$,
- σ and σ' are identical modulo $\text{EQ}(\mathcal{T}, \hat{A})$, and
- $\text{EQ}(\mathcal{T}, \hat{A}) \subseteq \text{APPLY}_{\text{SMP}}(T, \hat{A}, e, \theta)$,

we can infer that $T, \text{APPLY}_{\text{SMP}}(T, \hat{A}, e, \theta) \models \alpha$. Hence the claim holds. □

By combining Lemma 6.2 and Lemma 6.4, we get that $do()$ on $\mathcal{K}$ and $\text{DO}_{\text{SMP}}()$ on $\hat{\mathcal{K}}$ behave equivalently, when starting from equivalent ABoxes.

Lemma 6.5. *If* $A_1 \equiv_{T,\text{ALPH}(\mathcal{K})} A_2$ *then* $do(\mathcal{T}, A_1, \gamma\theta) \equiv_{T,\text{ALPH}(\mathcal{K})} \text{DO}_{\text{SMP}}(\mathcal{T}, A_2, \hat{\gamma}\theta)$.

Proof. The claim is a direct consequence of Lemma 6.2, Lemma 6.4, the equivalence between A_1 and A_2, and the observation that logical equivalence is transitive. □

Given a KAB $\mathcal{K}$ and its normalized version $\hat{\mathcal{K}}$, we call the transition system generated in the same way as $\mathcal{K}$, but using $\text{DO}_{\text{SMP}}()$ on $\hat{\mathcal{K}}$ instead of $do()$ on $\mathcal{K}$, the *normalized transition system* generated by $\hat{\mathcal{K}}$, and denote it with $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$.

Lemma 6.6. *Given a KAB* $\mathcal{K}$*, the transition systems* $\mathcal{K}$ *and* $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$ *are bisimilar.*

Proof. Let $\mathcal{K} = (\mathbb{U}, T, \Sigma, s_0, abox, \Rightarrow)$ and $\Upsilon_{\hat{\mathcal{K}},\text{SMP}} = (\mathbb{U}, T, \Sigma_{\text{SMP}}, s_0, abox_{\text{SMP}}, \Rightarrow_{\text{SMP}})$. We define the relation $\mathcal{B} \subseteq \Sigma \times \Sigma_{\text{SMP}}$ as follows: $(s_1, s_2) \in \mathcal{B}$ iff $abox(s_1) \equiv_{T,\text{ALPH}(\mathcal{K})} abox_{\text{SMP}}(s_2)$ and show that $\mathcal{B}$ is a bisimulation. To do so, we prove that $\mathcal{B}$ is closed under the definition of bisimulation itself. Indeed, if $(s_1, s_2) \in \mathcal{B}$, then:

- $abox(s_1) \equiv_{T,\text{ALPH}(\mathcal{K})} abox(s_2)$ by definition.
- If $s_1 \Rightarrow s'_1$ then there exists an action γ and a substitution θ such that $s'_1 = do(T, abox(s_1), \gamma\theta)$ (notice that $abox(s_1) = s_1$) and s'_1 is consistent w.r.t. $\mathcal{T}$. Now let us consider $s'_2 = \text{DO}_{\text{SMP}}(T, abox(s_2), \hat{\gamma}\theta)$. Since $abox(s_1) \equiv_{T,\text{ALPH}(\mathcal{K})} abox(s_2)$, then by Lemma 6.5, we have $s'_1 \equiv_{T,\text{ALPH}(\mathcal{K})} s'_2$. Therefore, s'_2 is consistent w.r.t. T, and hence $s_2 \Rightarrow_{\text{SMP}} s'_2$, and $(s'_1, s'_2) \in \mathcal{B}$.
- Similarly, if $s_2 \Rightarrow_{\text{SMP}} s'_2$ then there exists an action $\hat{\gamma}$ and a substitution θ such that $s'_2 = \text{DO}_{\text{SMP}}(T, abox(s_2), \hat{\gamma}\theta)$ and s'_2 is consistent w.r.t. $\mathcal{T}$. Now let us consider $s'_1 = do(T, abox(s_1), \gamma\theta)$. Since $s_2 \equiv_{T,\text{ALPH}(\mathcal{K})} s_1$, then by by Lemma 6.5, we have $s'_2 \equiv_{T,\text{ALPH}(\mathcal{K})} s'_1$ Therefore, s'_1 is consistent w.r.t. $\mathcal{T}$, and hence $s_1 \Rightarrow s'_1$, and, considering that equivalence enjoys symmetry, we have $(s'_1, s'_2) \in \mathcal{B}$.

This proves the claim. □

The direct consequence of the above lemma is that, by considering the Bismulation Invariance Theorem 6.3, we can faithfully check $\mu\mathcal{L}_A$ formulas over $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$ instead of $\mathcal{K}$.

6.5.3 Positive dominant

Our next step is to show that for a weakly acyclic KAB $\mathcal{K}$, the normalized transition system $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$ is finite. We do so by considering another transition system, which is behaviourally unrelated to $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$, and hence to $\mathcal{K}$, but whose ABoxes bound the ABoxes of $\Upsilon_{\hat{\mathcal{K}},\text{SMP}}$. We obtain such a transition system essentially by ignoring all negative information and equalities. This allows us to refer back to the literature on data exchange to show boundness. We call such transition system *positive dominant*.

Given a normalized KAB $\hat{\mathcal{K}} = (\mathcal{T}, \hat{A}_0, \hat{\Gamma}, \Pi)$, we define the *positive dominant* of $\mathcal{K}$ as the KAB

$$\mathcal{K}^+ = (\emptyset, \hat{A}_0^{\not\in\mathbb{Q}}, \{\gamma^+\}, \{\texttt{true} \mapsto \gamma^+\}).$$

The only action γ^+ is without parameters and its effect specification is constituted by *CopyAll*, and by one effect of the form

$$[q_i^{++}] \rightsquigarrow A_i'^{\not\in\mathbb{Q}}$$

for each effect $[q_i^{++}] \wedge q_i^{=} \wedge Q_i^{-} \rightsquigarrow A_i'$ in every action of $\hat{\Gamma}$. Observe that the parameters of the actions in $\hat{\Gamma}$ become simply free variables in γ^+.

Notice that γ^+ is applicable at every step because the process trivially always allows it. The resulting state is always consistent, since $\mathcal{K}^+$ has an empty TBox. Moreover, no equality assertion is ever generated. The transition system $\mathcal{K}^+$ is constituted by a single run, which incrementally accumulates all the facts that can be derived by the iterated application of γ^+ over such increasing ABox. This behaviour closely resembles the chase of TGDs in data exchange, where an application of γ^+ corresponds to a "parallel" chase step [58].

From a technical point of view, notice that $\mathcal{K}^+$ is already in normalized form (i.e., $\mathcal{K}^+ = \hat{K^+}$), and that $do()$ and $\text{DO}_{\text{SMP}}()$ are identical since neither equality nor negation are considered. Hence $\mathcal{K}^+ = \Upsilon_{\hat{\mathcal{K}^+},\text{SMP}}$.

The next lemma shows that $\mathcal{K}^+$ preserves weak acyclicity of $\hat{\mathcal{K}}$.

Lemma 6.7. *If $\hat{\mathcal{K}}$ is weakly acyclic then also its positive dominant $\mathcal{K}^+$ is weakly acyclic.*

Proof. The claim follows from the fact that, by construction, the dependency graph $\mathcal{G}^+$ of $\mathcal{K}^+$ is equal to $\hat{\mathcal{G}}$. Indeed, both q_i^{++} and its connection with $\hat{A}_i$ are preserved by $\mathcal{K}^+$. Hence, we get the claim. □

Next we show that if $\mathcal{K}^+$ is weakly acyclic the active domain of the ABoxes in its transition system $\mathcal{K}^+$ are polynomially bounded by the active domain of the initial ABox.

Lemma 6.8. *If $\mathcal{K}^+$ is weakly acyclic, then there exists a polynomial $\mathcal{P}(\cdot)$ such that*

$$|\text{ADOM}(\mathcal{K}^+)| < \mathcal{P}(|\text{ADOM}(\hat{A}_0^{\not\in\mathbb{Q}})|).$$

Proof. We observe that there exists a strict connection between the execution of $\mathcal{K}^+$ and the chase of a set of TGDs in data exchange. Therefore, the proof closely resembles the one of [65, Theorem 3.9], where it is shown that for weakly acyclic TGDs, every chase sequence is bounded.

Let $\mathcal{K}^+ = (\mathbb{U}, \emptyset, \Sigma, {A_0}^{\not\mathbb{Q}}, abox, \Rightarrow)$, let $\mathcal{G}^+ = (V, E)$ be the dependency graph of $\mathcal{K}^+$, and let $n = |\textsc{adom}({A_0}^{\not\mathbb{Q}})|$. For every node $p \in V$, we consider an *incoming path* to be any (finite or infinite) path ending in p. Let $rank(p)$ be the maximum number of special edges on any such incoming path. Since $\mathcal{K}^+$ is weakly acyclic by hypothesis, $\mathcal{G}^+$ does not contain cycles going through special edges, and therefore $rank(p)$ is finite. Let r be the maximum among $rank(p_i)$ over all nodes. We observe that $r \leq |V|$; indeed no path can lead to the same node twice using special edges, otherwise $\mathcal{G}^+$ would contain a cycle going through special edges, thus breaking the weak acyclicity hypothesis. Next we observe that we can partition the nodes in V according to their rank, obtaining a set of sets $\{V_0, V_1, \ldots, V_r\}$, where V_i is the set of all nodes with rank i.

Let us now consider a state A obtained from ${A_0}^{\not\mathbb{Q}}$ by applying the only action γ^+ contained in $\mathcal{K}^+$ an arbitrary number of times. We now prove, by induction on i, the following claim: for every i there exists a polynomial $\mathcal{P}_i$ such that the total number of distinct values c that occur in A at positions in V_i is at most $\mathcal{P}_i(n)$.

(Base case) Consider $p \in V_0$. By definition, p has no incoming path containing special edges. Therefore, no new values are stored in p along the run ${A_0}^{\not\mathbb{Q}} \Rightarrow \cdots \Rightarrow A$. Indeed p can just store values that are part of the initial ABox ${A_0}^{\not\mathbb{Q}}$. This holds for all nodes in V_0, and hence we can fix $\mathcal{P}_0(n) = n$.

(Inductive step) Consider $p \in V_i$, with $i \in \{1, \ldots, r\}$. The first kind of values that may be stored inside p are those values that were stored inside p itself in ${A_0}^{\not\mathbb{Q}}$. The number of such values is at most n. In addition, a value may be stored in p for two reasons: either it is copied from some other position $p' \in V_j$ with $i \neq j$, or it is generated as a possibly new Skolem term, built when applying effects that contain a Skolem function in their head.

We first determine the number of fresh individuals that can be generated from Skolem terms. The possibility of generating and storing a new value in p as a result of an action is reflected by the presence of special edges. By definition, any special edge entering p must start from a node $p' \in V_0 \cup \cdots \cup V_{i-1}$. By induction hypothesis, the number of distinct values that can exist in p' is bounded by $\mathcal{H}(n) = \sum_{j \in \{0,\ldots,i-1\}} \mathcal{P}_j(n)$. Let b_a be the maximum number of special edges that enter a position, over all positions in the TBox; b_a bounds the arity taken by each Skolem term contained in γ. Then for every choice of b_a values in $V_0 \cup \cdots \cup V_{i-1}$ (one for each special edge that can enter a position), the number of new values generated at position p is bounded by $t_f \cdot \mathcal{H}(n)^{b_a}$, where t_f is the total number of facts contained in all effects of γ^+. Note that this number does not depend on the data in ${A_0}^{\not\mathbb{Q}}$. By considering all positions in V_i, the total number of values that can be generated is then bounded by $\mathcal{F}(n) = |V_i| \cdot t_f \cdot \mathcal{H}(n)^{b_a}$. Clearly, $\mathcal{F}(\cdot)$ is a polynomial, because t_f and b_a are determined by γ^+.

We count next the number of distinct values that can be copied to positions of V_i from positions of V_j, with $j \neq i$. A copy is represented in the graph as a normal edge going from a node in V_j to a node in V_i, with $j \neq i$. We observe first that such normal edges can start only from nodes in $V_0 \cup \cdots \cup V_{i-1}$, that is, they cannot start from nodes in V_j with $j > i$. We prove this by contradiction. Assume that there exists $p' \to p \in E$, such that $p \in V_i$ and $p' \in V_j$ with $j > i$. In this case, the rank of p would be $j > i$, which contradicts the fact that $p \in V_i$. As a consequence, the number of distinct values that can be copied to positions in V_i is bounded by the total number of values in $V_0 \cup \cdots \cup V_{i-1}$, which corresponds to $\mathcal{H}(n)$ from our previous consideration.

Putting it all together, we define $\mathcal{P}_i(n) = n + \mathcal{F}(n) + \mathcal{H}(n)$. Since $\mathcal{P}_i(\cdot)$ is a polynomial, the claim is proven.

Notice that, in the above claim, i is bounded by r, which is a constant. Hence, there exists a fixed polynomial $\mathcal{P}(\cdot)$ such that the number of distinct values that can exist in every state $s \in \Sigma$ is bounded by $\mathcal{P}(n)$. $\mathcal{K}^+$ is inflationary, because when γ^+ is applied it copies all concept and role assertions from the current to the next state. Since $\mathcal{K}^+$ contains only a single run, $\mathcal{P}(n)$ is a bound for $\textsc{adom}(\mathcal{K}^+)$ as well. □

The key feature of the positive dominant $\mathcal{K}^+$ is that it gives an overestimation of the ABoxes generated by the normalized KAB $\hat{\mathcal{K}}$.

Lemma 6.9. $\textsc{adom}(\hat{\mathcal{K}}) \subseteq \textsc{adom}(\mathcal{K}^+)$

Proof. Let $\hat{\mathcal{K}} = (T, \hat{A}_0, \hat{\Gamma}, \Pi)$ and $\mathcal{K}^+ = (\emptyset, \hat{A}_0^{\not\Leftrightarrow Q}, \{\gamma^+\}, \{\texttt{true} \mapsto \gamma^+\})$.

We first observe that, for every ABox A in $\hat{\mathcal{K}}$, $\textsc{adom}(A) = \textsc{adom}(A^{\not\Leftrightarrow Q})$, by definition of $\hat{\mathcal{K}}$ (this is the role of the special concept *Dummy*).

We show by induction on the construction of $\hat{\mathcal{K}}(\mathbb{U}, \mathcal{T}, \Sigma_1, \hat{A}_0, abox, \Rightarrow_1)$ and $\mathcal{K}^+ = (\mathbb{U}, \emptyset, \Sigma_2, \hat{A}_0^{\not\Leftrightarrow Q}, abox, \Rightarrow_2)$, that for each state $A_1 \in \Sigma_1$ we have that there exists a state $A_2 \in \Sigma_2$ such that $A_1^{\not\Leftrightarrow Q} \subseteq A_2$.

The base case holds for the initial states $\hat{A}_0$ and $\hat{A}_0^{\not\Leftrightarrow Q}$ of the two transition systems by definition. For the inductive case, we have to show that, given $A_1 \in \Sigma_1$ and $A_2 \in \Sigma_2$ with $A_1^{\not\Leftrightarrow Q} \subseteq A_2$, for each $A_1' \in \Sigma_1$ with $A_1 \Rightarrow_1 A_1'$, the unique state $A_2' \in \Sigma_2$ with $A_2 \Rightarrow_2 A_2'$ is such that $A_1' \subseteq A_2'$. To show this, note that $A_1 \Rightarrow_1 A_1'$ if there exists an action γ of $\hat{\mathcal{K}}$ and a substitution θ for the parameters of γ such that $A_1' = \textsc{do}_{\textsc{smp}}(T, A_1, \gamma\theta)$. Similarly, taking into account that γ^+ has no parameters and is always executable in $\mathcal{K}^+$, we have that $A_2' = do(T, A_2, \gamma^+) = \textsc{do}_{\textsc{smp}}(T, A_2, \gamma^+)$. By construction of $\mathcal{K}^+$, for each effect $e_1 \in \gamma$ of the form

$$e_1 : [q^{++}] \wedge q^{=} \wedge Q^{-} \rightsquigarrow A'_{e_1},$$

there is an effect $e_2 \in \gamma^+$ of the form

$$e_2 : [q^{++}] \rightsquigarrow A'^{\not\Leftrightarrow Q}_{e_1},$$

where $A'^{\not\Leftrightarrow Q}_{e_1}$ is obtained from A'_{e_1} by removing all equality assertions. By induction hypothesis, we have that $A_1^{\not\Leftrightarrow Q} \subseteq A_2$. By observing that $\textsc{ans}([q^{++}]\theta, \emptyset, A_1^{\not\Leftrightarrow Q}) \cap \textsc{ans}((q^{=} \wedge Q^{-})\theta, \emptyset, A_1) \subseteq \textsc{ans}([q^{++}], \emptyset, A_2)$, we then obtain that $A'^{\not\Leftrightarrow Q}_{e_1} \subseteq A'_{e_2}$, where $A'_{e_1} = \textsc{apply}_{\textsc{smp}}(T, A_1, e_1, \theta)$ and $A'_{e_2} = \textsc{apply}(\emptyset, A_2, e_2, \emptyset)$. Hence, we get the claim that $A_1'^{\not\Leftrightarrow Q} \subseteq A_2'$.

Now since for an ABox A of $\hat{\mathcal{K}}$ the active domain $\textsc{adom}(A)$ of A and $\textsc{adom}(A^{\not\Leftrightarrow Q})$ are identical by construction, and since $\textsc{adom}(\hat{\mathcal{K}})$ and $\textsc{adom}(\mathcal{K}^+)$ are simply the union of the active domains of all generated ABoxes, we get the claim. □

6.5.4 Putting it all together

If a KAB $\mathcal{K}$ is weakly acyclic, then, by Lemma 6.3, its normalized form $\hat{\mathcal{K}}$ is weakly acyclic as well and, by Lemma 6.7, so is its positive dominant $\mathcal{K}^+$. Hence, by

Lemma 6.8, the size of the active domain $\textsc{adom}(\mathcal{K}^+)$ of the transition system $\mathcal{K}^+$ of $\mathcal{K}^+$ is polynomially related to the size of its initial ABox.

Now, by Lemma 6.9, this implies that also the size of the active domain $\textsc{adom}(\Upsilon_{\hat{\mathcal{K}},\text{SMP}})$ of the transition system $\hat{\mathcal{K}}$ of $\hat{\mathcal{K}}$ is polynomially related to the size of its initial ABox. Hence, the number of possible states of $\hat{\mathcal{K}}$ is finite, and in fact at most exponential in the size of the initial ABox. It follows that checking $\mu\mathcal{L}_A$ formulae over $\hat{\mathcal{K}}$ can be done in ExpTime w.r.t. the size of $\mathcal{K}$.

Finally, by Lemma 6.6, $\hat{\mathcal{K}}$ and $\mathcal{K}$ are bisimilar, and by the Bismulation Invariance Theorem 6.3, $\hat{\mathcal{K}}$ and $\mathcal{K}$ satisfy exactly the same $\mu\mathcal{L}_A$ formulae. Hence, to check a $\mu\mathcal{L}_A$ formula on $\mathcal{K}$ it is sufficient to check it over $\hat{\mathcal{K}}$, which can be done in ExpTime. This concludes the proof of Theorem 6.5. □

6.6 Discussion

The framework presented in this Chapter substantially differs from the ones explored before. Indeed both Chapter 2 and 3 rely on a purely relational setting which leads to an ad-hoc interpretation of equality, where each null value/Skolem term is considered only equal to itself. Here instead, we allow for sophisticated schema constraints, i.e., the TBox itself, and provide at the same time a more fine-grained treatment of equality, where individuals can be inferred to be equal due to the application of such schema constraints and/or the execution of some action.

Merging temporal verification with ontologies, processes and actions positions this work in an area that is influenced by different fields. Therefore related work range from AI to BPM by passing through knowledge representation. Given that we already discussed the BPM and verification literature, here we focus on knowledge representation and reasoning about actions.

Combining description logics and temporal logics. Work addressing the combination of description logics and temporal logics can be found in [121, 139, 70], where a logics with two-dimensional semantics, one for time and the other for the DL domain, is developed. Unfortunately the possibility of specifying that roles preserve their extension over time (the so called *rigid roles*) causes undecidability [119, 135]. One way to regain decidability is by allowing temporal operators only on TBox and ABox assertions [101, 7] and by fixing apriori the individual terms mentioned in the ABox. The restriction on the language is tightly related to the functional approach adopted here (we indeed allow temporal operators in fronts of assertions only) although we do not fix the individuals, as we possibly introduce new Skolem terms.

Combining description logics and actions. Combinations of description logics and action theories again at the level of models have been studied in [98, 97], but only w.r.t. the two classical problems in reasoning about actions, namely *projection* and *executability*. More sophisticated temporal properties lead to undecidability.

Possibly the first proposal based implicitly on the functional view of the KB was the pioneering work in [52], where an epistemic description logic (based on certain

answers) combined with an action formalism is adopted to describe routines of a mobile robot. An important point there is that individual terms are bounded and fixed a priori.

The functional view approach was first spelled out in [33, 34]. In that work, only projection and executability are studied, however there is a distinction beween the KB in the states and the actions (there specified as updates), so that the framework gives rise to a single transition system whose states are labeled by KBs (in fact the TBox is fixed while the ABox changes from state to state). However, again, the individual terms considered are fixed a priori and hence the resulting transition system is finite.

An interesting alternative way to combine description logics and reasoning about actions is the one reported in [78]. There, a description logics KB is used as a special FOL theory describing the initial situation in a situation calculus basic action theory [118]. Notice that as a result, TBox assertions do not act as state constraints [95], which would lead to undecidability as discussed above [139, 70]; in fact they essentially do not persist in any way through actions.

Chapter 7

Runtime Verification

We investigated techniques to verify an artifact systems offline. We showed that such a problem is undecidable in general, and several restrictions should be put in place to achieve decidability. In many practical scenarios, however, it is not necessary to verify all possible executions of the system offline, but just a specific execution, e.g., the current one. Let us assume, for instance, that there is not even a formal description of the system to be analyzed, e.g., because unknown, or because highly unstructured (hence an offline analysis cannot be performed) but still, there are underlying data changing and temporal properties over such data should be enforced.

A representative example is health care. Governments have general "guidelines" that do not describe a precise process, but rather a set of cases in which some activities have to be performed. As an example, when a head injury happens, the patient may be rushed to a hospital. During the transportation and upon arrival in emergency room, the vitals, Glasgow coma scale (GCS), injury severity score (ISS) and other test results are measured. *Physicians make decisions on diagnostics and treatments based on collected data and "protocols".* Treatment protocols are formulated based on analysis of past patient records and change often. For example, a protocol may state that lightly injured (low ISS score) patients over 65 years old with decreasing GCS scores should have additional tests. After analyzing data, the condition in the protocol may be revised to lightly injured patients over 79 with decreasing GCS or increasing heart rate. In such a scenario, *runtime monitoring of data would help physicians to adhere to the protocol and assists their decisions.*

In this Chapter we focus on the support for runtime enforcement of first-order LTL formulas for data-centric process executions and study the problem of how to incrementally evaluate them.

Technically, properties are specified in a first-order language extended with (future time) linear-time logic (LTL) operators and we check them on a sequence of relational data instances incrementally. This treatment reflects artifact-centric systems, and we believe that such techniques provide a practical and effective alternative to offline verification.

Our approach is inspired by works on incremental evaluation of queries [59, 38], where the key idea to incrementally evaluate a temporal query is to store data (from the history) in "views" relevant to the query in question. The query can be

answered using the stored auxiliary view results during execution without accessing the history. When the system execution advances to a new snapshot, new results of views are computed from querying the current and new snapshots, and the current views, without using temporal operators. It is important to note that, since the incremental step needs no history, it can be effectively computed (e.g., by non-temporal queries). This is very practical.

We initiate the study on incremental evaluation of first-order temporal properties over data instances evolving over time, by proposing an automata-based approach: we extend the runtime verification technique presented in [14] to a first-order setting by constructing a first-order Büchi automaton. Such an automaton, along with some auxiliary data structure evolving together with the data evolution, is able to monitor the property in an incremental fashion.

More generally we provide an alternative way of performing formal verification of artifact-centric models and other business process models.

7.1 First-order LTL

We assume the data schema to be relational. We define the *data schema* as a tuple $\mathcal{S} = (R_1 \dots R_n, \Delta)$ where $R_1 \dots R_n$ are relation symbols with an associated arity and Δ is a *fixed* a-priori and *finite* set of constants. An instance I of $\mathcal{S}$ interprets each relation symbol R_i with arity n as a relation $R_i^I \subseteq \Delta^n$. Values in Δ are interpreted as themselves, blurring the distinctions between constants and values. We use the terms instance, snapshot or interpretation interchangeably. Given a schema $\mathcal{S}$, we use the symbol $\mathcal{I}$ to denote all possible interpretations for $\mathcal{S}$.

The analysis we perform consists in checking temporal properties while data evolve. In particular, we provide the theoretical foundations for building a module that takes in input a first-order temporal property and, each time data changes in the current instance, it inspects the new instance and checks whether the temporal property is verified, falsified or not satisfied nor falsified yet. If the property is falsified, the new instance is rejected, and the execution must continue starting from the previous one, i.e., we rollback to the previous state. Otherwise, the new instance is accepted. In such a scenario we need to recognize a violation at the earlier possible time, in order to be sure that before the last update, i.e., from the previous snapshot, there exists a possible execution that satisfies the formula.

We present a first-order LTL language that merges the capabilities of first-order logic for querying the instance with the LTL temporal operators.

Definition 7.1 (First-order LTL Language $\mathcal{L}$ Syntax)**.** *Given a data schema $\mathcal{S}$, the set of* closed *first-order LTL formulas Φ of our language $\mathcal{L}$ are built with the following syntax:*

$$\begin{aligned} \Phi^\ell &:= \mathit{true} \mid \mathit{Atom} \mid \neg\Phi^\ell \mid \Phi_1^\ell \wedge \Phi_2^\ell \mid \forall x.\Phi^\ell \\ \Phi^t &:= \Phi^\ell \mid \boldsymbol{X}\ \Phi^t \mid \Phi_1^t\ \boldsymbol{U}\ \Phi_2^t \mid \neg\Phi^t \mid \Phi_1^t \wedge \Phi_2^t \\ \Phi &:= \Phi^t \mid \neg\Phi \mid \forall x.\Phi \end{aligned}$$

where x is a variable symbol and Atom is an atomic first-order formula or atom, i.e., a formula inductively defined as follows: true is an atomic formula; if t_1 and

t_2 are constants in Δ or variables, then $t_1 = t_2$ is an atomic formula and if $t_1 \dots t_n$ are constants or variables and R a relation symbol of arity n, then $R(t_1 \dots t_n)$ is an atomic formula. Since Φ is closed, we assume that all variables symbols are in the scope of a quantifier.

We call *local* formulas the set Φ^ℓ because they do not include any temporal operators. A formula in Φ^ℓ is a first-order with equality but no function symbols. They express a local constraint, and therefore they can be checked by looking at a single snapshot.

The set of formulas in Φ^t are *temporal* formulas, and therefore they include $\mathbf{X}$ and $\mathbf{U}$ logic symbols that are the classical LTL *next* and *until* operators. Satisfiability of temporal formulas cannot be established by looking at a single snapshot only. However, they do not include any quantifier that spans over different snapshots.

Finally, the set Φ is made up by formulas that have quantifiers for variables that occur in the scope of temporal operators. We call such variables *across-state* existential and universal variables. Such formulas are hard and costly to be monitored, because, in general, they require the whole history of snapshots seen so far to establish whether they are verified.

Notice that the scope of the quantifiers is required to be the entire formula. In other words, our language is not full first-order LTL, since we require all across-state variables to appear in the front of the formula. As an example, $\forall x.(R_1(x)\ \boldsymbol{U}\ (\neg\forall y.(R_2(x,y) \wedge\ \boldsymbol{X}\ R_3(y))))$ is not allowed, since variable y is across-states (because in the scope of $\boldsymbol{X}$) but its quantifier is not in the front. The distinction between local and temporal formulas is central to our approach.

The logic symbols $\vee$, $\boldsymbol{F}$, $\boldsymbol{G}$ and $\exists$ defined as follows: $\Phi_1 \vee \Phi_2 := \neg(\neg\Phi_1 \wedge \neg\Phi_2)$; $\exists x.\Phi := \neg\forall x.\neg\Phi$; $\boldsymbol{F}\ \Phi := \mathit{true}\ \boldsymbol{U}\ \Phi$; $\boldsymbol{G}\ \Phi := \neg\ \boldsymbol{F}\ \Phi$.

Every formula in $\mathcal{L}$ can be translated in an equivalent formula in *prenex normal form*, i.e., with all quantifier in the front. From now on we assume formulas to be in such a form.

Before showing the semantics of $\mathcal{L}$, we need to introduce the notion of assignment. Let I be a first-order interpretation, i.e., a snapshot for $\mathcal{S}$, an *assignment* η is a function that associates to each free variable x a value $\eta(x)$ in Δ. Let η be an assignment, then $\eta_{x/d}$ is the assignment that agrees with η except for the value $d \in \Delta$ that is now assigned to the variable x. We denote by $\Phi[\eta]$ is the formula obtained from Φ by replacing variables symbols with values in η.

Let us now define the semantics of our language. Our analysis is performed at runtime, and hence its semantics is based on finite-length executions or paths. Such a semantics is defined starting from the classical infinite-length paths semantics, therefore we first show the latter, and then we turn to the former. An infinite *path* is an infinite sequence of snapshots $I_0, I_1 \dots$, i.e., given a schema $\mathcal{S}$ it is a function $\pi : \mathbb{N} \to \mathcal{I}$ that assigns a snapshot to each time instant i.

Definition 7.2 (Infinite path $\mathcal{L}$ Semantics)**.** *Given a formula $\Phi \in \mathcal{L}$ over a schema $\mathcal{S}$, an assignment η, a path π and an instant of time i we have that:*

- $(\pi, i, \eta) \models \Phi^\ell$ *iff* $(\pi(i), \eta) \models \Phi^\ell$*, where* $(\pi(i), \eta) \models \Phi^\ell$ *is the classical first-order logic evaluation function;*

- $(\pi, i, \eta) \models \boldsymbol{X}\ \Phi^t$ *iff* $(\pi, i+1, \eta) \models \Phi^t$;
- $(\pi, i, \eta) \models \Phi_1^t\ \boldsymbol{U}\ \Phi_2^t$ *iff for some* $j \geq i$ *we have that* $(\pi, j, \eta) \models \Phi_2^t$ *and for all* $i \leq k < j$ *we have that* $(\pi, k, \eta) \models \Phi_1^t$;
- $(\pi, i, \eta) \models \neg\Phi$ *iff* $(\pi, i, \eta) \not\models \Phi$
- $(\pi, i, \eta) \models \Phi_1 \wedge \Phi_2$ *iff* $(\pi, i, \eta) \models \Phi_1$ *and* $(\pi, i, \eta) \models \Phi_2$;
- $(\pi, i, \eta) \models \forall x.\Phi$ *iff for all* $d \in \Delta$ *we have* $(\pi, i, \eta_{x/d}) \models \Phi$;

Further, $(\pi, \eta) \models \Phi$ *iff* $(\pi, 0, \eta) \models \Phi$ *and, since every formula in* $\mathcal{L}$ *has no free variables, we can simply write* $\pi \models \Phi$.

Some remarks are in order. First of all, note that when a formula does not contain any temporal operators, i.e., when it is local, its semantics corresponds exactly to the classical first-order semantics. Indeed, in order to evaluate a local formula, we do not need the whole path, but the current snapshot only. Moreover, the domain Δ is the same for each instant of time (see [68] for a dissertation on different semantics for first-order modal logics).

We now turn to the finite-path semantics. Since the first introduction of LTL by Pnueli [115], several different semantics for finite-path LTL has been proposed, see e.g., [102, 60, 48]. Here we adapt the one in [14] to our first-order setting. Such a semantics is strongly related with the notion of *bad prefixes*, that has been introduced in [90]. Given a formula Φ in $\mathcal{L}$, a *bad prefix* for Φ is a finite path such that any infinite extension of it does not satisfy Φ. In other words, no matter the continuation of the prefix, the formula Φ will be always evaluated to false. As an example, safety properties such as "always p holds" always have a bad prefix that violates them, that is, a finite path that contains a state where p does not hold. Analogously, a *good prefix* can be defined as a finite path that, no matter its continuation, it will always satisfy the property Φ. Eventualities, such as "eventually p holds", have always a finite path that satisfies them. There are several LTL properties that cannot be satisfied nor falsified by any finite trace, e.g., "infinitely often p holds". It is easy to see that any finite path can be extended to an infinite one satisfying the formula as well as falsifying it. Such formulas are called in [14] *non-monitorable*.

Definition 7.3 (Finite path $\mathcal{L}$ Semantics)**.** *Given a formula* Φ *over a schema* $\mathcal{S}$ *and a finite path of length* k $\pi[k]$*, the truth value of a formula* Φ *on* $\pi[k]$*, denoted by* $[\pi[k] \models \Phi]$*, is an element of the set* $\mathbb{B}_3 = \{true, false, ?\}$ *defined as follows:*

- $[\pi[k] \models \Phi] := true$ *iff* $\pi[k]$ *is bad prefix for* $\neg\Phi$;
- $[\pi[k] \models \Phi] := false$ *iff* $\pi[k]$ *is bad prefix for* Φ;
- $?$ *otherwise.*

Notice that a bad prefix for $\neg\Phi$ is a good prefix for Φ. The core technical issue of our problem can now be re-formulated as recognizing the bad and good prefixes. Indeed, when a new snapshot I_i is presented as input, we have to check whether $I_0 \dots I_i$ is a bad or good prefix or neither of the two, i.e., we have to compute the relation $[I_0 \dots I_i \models \Phi]$. In the classical, propositional version, the problem of recognizing a bad prefix for a propositional formula ψ can be solved by building a

monitor for ψ (see, e.g., [14, 48]). The procedure is centered on the construction of a Büchi automaton for ψ (e.g., following the procedure in [74]) that represents the (infinite strings) language $L(\psi)$. A Büchi automaton is an automaton on infinite strings, whose acceptance condition requires that a final state is visited infinitely often.

We report here the automaton construction for an LTL formula in [12] and others notions used for its definition.

Let Ψ be a propositional formula over the alphabet AP. The closure $cl(\Psi)$ consists in all subformulas ψ of Ψ and their negation $\neg\psi$.

Definition 7.4 (Elementary set of formulas)**.** *$B \subseteq cl(\Psi)$ is elementary if:*

1. *B is consistent with respect to propositional logic, i.e., for all $\phi_1 \wedge \phi_2$, $\psi \in cl(\Psi)$:*
   - *$\phi_1 \wedge \phi_2 \in B \leftrightarrow \phi_1 \in B$ and $\phi_2 \in B$;*
   - *$\psi \in B \rightarrow \neg\psi \notin B$;*
   - *$\texttt{true} \in cl(\hat{\Phi}) \rightarrow true \in B$.*
2. *B is* locally consistent *with respect to the until operator, i.e., for all $\phi_1 \ \boldsymbol{U} \ \phi_2 \in cl(\Psi)$:*
   - *$\phi_2 \in B \rightarrow \phi_1 \ \boldsymbol{U} \ \phi_2 \in B$;*
   - *$\phi_1 \ \boldsymbol{U} \ \phi_2 \in B$ and $\phi_2 \notin B \rightarrow \phi_1 \in B$.*
3. *B is* maximal*, i.e., for all $\psi \in cl(\Psi)$:*
   - *$\psi \notin B \rightarrow \neg\psi \in B$.*

Definition 7.5 (Büchi automaton for LTL formula)**.** *Given a propositional formula Ψ over AP, the Büchi automaton $\mathcal{A}$ such that $L(\Psi) = L(\mathcal{A})$ is a tuple $(2^{AP}, Q, \delta, Q_0, F)$ where:*

- *the set of states Q is the set of all elementary set of formulas $B \subseteq cl(\Psi)$;*
- *the set of initial state $Q_0 = \{B \in Q \mid \Psi \in B\}$;*
- *the set of final states $F = \{F_{\phi_1 \boldsymbol{U} \phi_2} \in cl(\Psi)\}$ where $F_{\phi_1 \boldsymbol{U} \phi_2} = \{B \in Q \mid \phi_1 \ \boldsymbol{U} \ \phi_2 \notin B \text{ or } \phi_2 \in B\}$.*
- *the transition relation $\delta : Q \times 2^{AP} \rightarrow 2^Q$ si given by:*
  - *if $a \neq B \cap AP$, then $\delta(B, a) = \emptyset$;*
  - *if $a = B \cap AP$, then $\delta(B, a)$ is the set of all elementary formulas B' satisfying:*
    1. *for every $\boldsymbol{X} \ \psi \in cl(\Psi) : \boldsymbol{X} \ \psi \in B \leftrightarrow \psi \in B'$ and*
    2. *for every $\phi_1 \ \boldsymbol{U} \ \phi_2 \in cl(\Psi) : \phi_1 \ \boldsymbol{U} \ \phi_2 \in B \leftrightarrow (\phi_2 \in B \vee (\phi_1 \in B \wedge \phi_1 \ \boldsymbol{U} \ \phi_2 \in B'))$.*

A *run* of $\mathcal{A}$ on an infinite word $\alpha = a_0, a_1 \ldots$ (or ω-word) is an infinite state sequence $r(0), r(1) \ldots$ where the following holds: *(i)* $r(0) = q_0$ and *(ii)* $r(i) \in \varrho(r(i-1), a_i)$ for $i \leq 1$ if $\mathcal{A}$ is nondeterministic or $r(i) = \varrho(r(i-1), a_i)$ for $i \leq 1$ if $\mathcal{A}$ is deterministic.

An infinite word α is accepted by $\mathcal{A}$ iff there exists a run $r(0), r(1) \ldots$ that visits one of the states in F infinitely often. In other words, α is accepted if the run $r(0), r(1) \ldots$ cycles in a set of states containing a final state.

Given a propositional formula Ψ, the monitor for Ψ is constructed as follows: (1) the automaton for Ψ is generated and (2) states of the automaton that do not satisfy the Büchi condition, i.e., from which a path that leads to a cycle containing an accepting state does not exist, are marked with "bad".

To monitor an execution of a system, it is enough to navigate the automaton's transitions as the instances are presented in input. When a bad state is reached, the last instance must be rejected. This because from each "bad" state there is no way to accept any infinite words belonging to $L(\Psi)$, hence Ψ is falsified.

Notice that the monitor outputs the truth value ? for states that are not marked, because from them there exists a path leading to the acceptance condition, but the formula can still be falsified later on. Indeed, by using the automaton for Ψ we can recognize the bad prefixes only, but not the good ones. To fully capture the three-valued semantics presented before, two automata have to be used: one for Ψ for recognizing the bad prefixes and one for $\neg\Psi$ recognizing the good ones.

We propose an approach that is grounded on the aforementioned technique but that introduces some novelties and optimizations needed when dealing with first-order properties.

Before entering into the details of our methodology, we point out that our first-order formulas can be translated into an equivalent propositional formula. Indeed, given that we do not have any function symbols, and finite domain, the first-order syntax is just a shortcut for the propositional one. Given a first-order formula $\Phi \in \mathcal{L}$, we can build an *equivalent* propositional formula.

To do so, we need, for technical reasons, a function f that, given a first-order interpretation I, it returns a (local) formula $\mathsf{f}(I)$ that describes the interpretation.

Lemma 7.1. *Let $\mathcal{I}$ be the set of all interpretations for $\mathcal{L}$. Given a finite domain Δ we can build a function $\mathsf{f} : \mathcal{I} \to \mathcal{L}$ that, given an interpretation $I \in \mathcal{I}$, returns a first-order ground and variable-free formula such that for each infinite path π, formula $\Phi \in \mathcal{L}$, assignment η and $i \in \mathbb{N}$, we have that $(\pi, i, \eta) \models \Phi$ iff $(\mathsf{f}(\pi), i, \eta) \models \Phi$, where $\mathsf{f}(\pi)$ is the natural extension of f to paths, i.e., it is the path obtained from π by applying function f at the first-order interpretation $\pi(i)$ for each time instant $i \in \mathbb{N}$.*

Proof. The formula returned by $\mathsf{f}(I)$ is the conjunction of all positive facts $R_i(d_1, \ldots, d_n)$ when $(d_1, \ldots, d_n) \in R_i^I$ and the conjunction of all negative facts $\neg R_i(d_1, \ldots, d_n)$ when $(d_1, \ldots, d_n) \notin R_i^I$, for all relation symbol $R_i \in \mathcal{L}$ and tuple $(d_1, \ldots, d_n) \in \Delta^n$. □

In what follows, we refer to [136] for the classical, i.e., on infinite paths, LTL propositional semantics and to [14] to the LTL_3 semantics, i.e., on finite propositional paths.

Lemma 7.2 (Propositionalization)**.** *Let $\mathcal{L}$ be the language defined before, $\mathcal{L}^p$ a propositional LTL language and Δ a* finite *domain. Then we can build a mapping $\mathsf{p} : \mathcal{L} \to \mathcal{L}^p$ from formulas of $\mathcal{L}$ into propositional formulas of $\mathcal{L}^p$ such that, given a formula $\Phi \in \mathcal{L}$, an infinite path π, and a finite path $\pi[k]$:*

- $\pi \models \Phi \equiv \mathsf{p}(\mathsf{f}(\pi)) \models \mathsf{p}(\Phi)$;
- $[\pi[k] \models \Phi] \equiv [\mathsf{p}(\mathsf{f}(\pi[k])) \models \mathsf{p}(\Phi)]$

where $\mathsf{p}(\mathsf{f}(\pi))$ *is the natural extension of* p *to paths, i.e., it is the path obtained from* $\mathsf{f}(\pi)$ *by applying function* p *at the first-order interpretation* $\mathsf{f}(\pi(i))$ *for each time instant* $i \in \mathbb{N}$.

Proof. The proof is constructive. Function p is inductively defined as follows:

- $\mathsf{p}(true) ::= true$;
- $\mathsf{p}(P(t_1 \dots t_n)) ::= a$ where a is a constant symbol and such that $\mathsf{p}(P(t_1 \dots t_n)) ::= a$ and $\mathsf{p}(P'(t'_1 \dots t'_n)) ::= a$ iff $P'(t'_1 \dots t'_n) = P(t_1 \dots t_n)$;
- $\mathsf{p}(t_1 = t_2) ::= \mathsf{p}(t_1) = \mathsf{p}(t_2)$;
- $\mathsf{p}(\neg\Phi) ::= \neg\mathsf{p}(\Phi)$;
- $\mathsf{p}(\Phi_1 \wedge \Phi_2) ::= \mathsf{p}(\Phi_1) \wedge \mathsf{p}(\Phi_2)$;
- $\mathsf{p}(\forall x.\Phi) ::= \bigwedge_{d \in \Delta} \mathsf{p}(\Phi(x/d))$ where $\Phi(x/d)$ is obtained from Φ with the substitution x/d;
- $\mathsf{p}(\boldsymbol{X}\Phi) ::= \boldsymbol{X}\mathsf{p}(\Phi)$;
- $\mathsf{p}(\Phi_1 \boldsymbol{U} \Phi_2) ::= \mathsf{p}(\Phi_1) \boldsymbol{U} \mathsf{p}(\Phi_2)$;

□

The formula Φ can be propositionalized by applying the definition of function p. Given that we are under finite domain assumption, the procedure is guarantee to terminate. Notice that there are several ways to map atoms to propositional symbols, hence there are several different propositionalization functions, all equivalent modulo propositional symbol renaming. In the rest of the chapter we assume to fix one of those, say function p.

Given a formula $\Phi \in \mathcal{L}$, the size of the formula $\mathsf{p}(\Phi)$ is exponential in the number of universal variables, hence, in the worst case, on the length of Φ. More precisely, its size is $|\Delta|^{|\Phi|}$.

7.2 First-order automaton

Every formula $\Phi \in \mathcal{L}$ can be propositionalized. Therefore it is easy to see that to monitor Φ we could first propositionalize it, obtaining $\mathsf{p}(\Phi)$, and then we could use existing techniques for monitoring propositional formulas. However, building a monitor requires the construction of the Büchi automaton for $\mathsf{p}(\Phi)$ (see e.g., [74] for the actual procedure) that is exponential in the size of the formula, which, in turn, is exponentially bigger than the original Φ. Given that the automaton construction is $c^{|\mathsf{p}(\Phi)|}$, where c is a constant, we have that the overall complexity is $c^{|\Delta|^{|\Phi|}}$, that is, exponential in the size of the domain and double exponential in the size of the formula.

In this Section we illustrate how to monitor a first-order formula in exponential space in the size of the formula. In order to do so, we make use of a first-order automaton plus some data structures. As it will be clear later on, the auxiliary

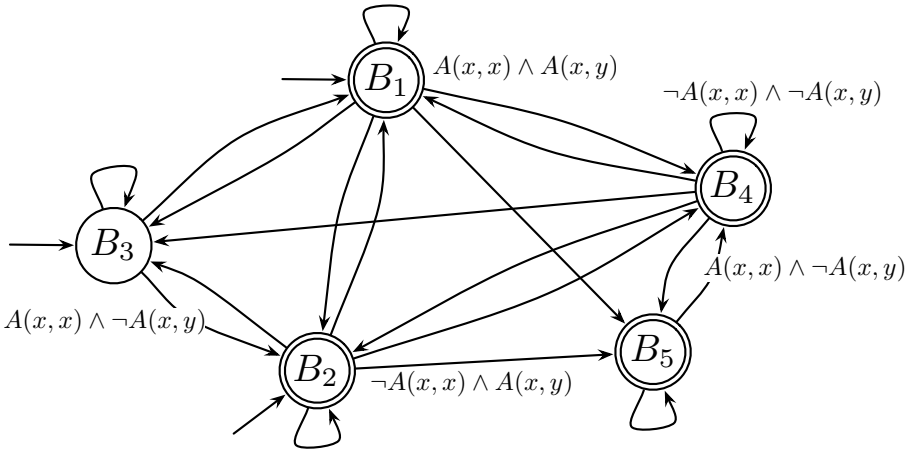


Figure 7.1. Graphical representation of the automaton $\mathcal{A}(\hat{\Phi}(x,y))$ in Example 7.2.1.

data structures are used to keep track of assignments to variables. The advantage of this methodology is to decouple the cost of building the automaton from the size of the domain.

Given a formula $\Phi \in \mathcal{L}$, in order to build the first-order automaton for Φ, we first drop all quantifiers from Φ, obtaining an open first-order formula $\hat{\Phi}$ and then we build the automaton for $\hat{\Phi}$. Indeed, given that $\hat{\Phi}$ contains no quantifiers, we can consider $Atom(\hat{\Phi})$ as propositional symbols and use a standard propositional procedure, e.g., the one in [12] for building the automaton for $\hat{\Phi}$. The formal procedure would require to first propositionalize the atoms of $\hat{\Phi}$, then build the automaton, and lastly use p^{-1} to translate back the propositional symbols into first-order formulas. To ease the presentation we skip the propositionalization step. Indeed, given that $\hat{\Phi}$ does not have quantifiers, function p (cf. Lemma 7.2) turns out to be a syntactic renaming of atoms.

The automaton $\mathcal{A}(\hat{\Phi})$ is likewise a propositional one, except for its transitions and states that are labeled with *open* first-order formulas.

Example 7.2.1. *Let* $\Phi(x,y) = \forall x,y.A(x,x) \ \boldsymbol{U}\ A(x,y)$. *Then* $\hat{\Phi} = A(x,x) \ \boldsymbol{U}\ A(x,y)$. *We follow the procedure in [12] illustrated before for the automaton construction. The closure of* $\hat{\Phi}$ *is* $cl(\hat{\Phi}) = \{A(x,x), \neg A(x,x), A(x,y), \neg A(x,y), A(x,x) \ \boldsymbol{U}\ A(x,y), \neg(A(x,x) \ \boldsymbol{U}\ A(x,y))\}$ *while the set of elementary formulas is:*

- $B_1 = \{A(x,x), A(x,y), A(x,x) \ \boldsymbol{U}\ A(x,y)\}$;
- $B_2 = \{\neg A(x,x), A(x,y), A(x,x) \ \boldsymbol{U}\ A(x,y)\}$;
- $B_3 = \{A(x,x), \neg A(x,y), A(x,x) \ \boldsymbol{U}\ A(x,y)\}$;
- $B_4 = \{\neg A(x,x), \neg A(x,y), \neg(A(x,x) \ \boldsymbol{U}\ A(x,y))\}$;
- $B_5 = \{A(x,x), \neg A(x,y), \neg(A(x,x) \ \boldsymbol{U}\ A(x,y))\}$.

The first order automaton for $\hat{\Phi}(x,y)$ *is the tuple* $\mathcal{A}(\hat{\Phi}(x,y)) = (2^{Atom(\hat{\Phi})}, Q, \delta, Q_0, F)$ *in Figure 7.1. Some labels on transitions have been omitted to keep the*

Figure readable, but they can easily be inferred: all outgoing edges of a state B_i have the same label, i.e., the formula characterizing $B_i \cap Atom(\hat{\Phi})$.

Our approach is grounded on the fact that a first-order automaton $\mathcal{A}(\hat{\Phi})$ along with some data structures, is capable of effectively simulating the propositional automata $\mathcal{A}(\mathsf{p}(\Phi))$ needed for monitoring $\mathsf{p}(\Phi)$. In the rest of the section we are going to show how to use $\mathcal{A}(\hat{\Phi})$ for runtime monitoring Φ according to the finite path semantics in Definition 7.3.

To ease the presentation, we proceed in two steps: we first illustrate how, given an assignment η for the variables in Φ, we can monitor the property $\Phi[\eta]$, where with $\Phi[\eta]$ we denote the formula obtained from Φ by ignoring the quantifiers and by replacing the variables with the assignment η. We then generalize the procedure by showing how to concurrently monitor all possible assignments using $\mathcal{A}(\hat{\Phi})$ and how to compose the results obtained for each assignment in order to evaluate the original formula.

Let us assume an assignment η for the variables of Φ. We now show a procedure that, given as input: *(i)* a first-order automaton $\mathcal{A}(\hat{\Phi})$; *(ii)* an assignment η for the variables and *(iii)* the snapshots (as data evolve), is able to recognize the bad prefixes for $\mathsf{p}(\Phi[\eta])$. The steps of the procedure follow:

1. we propositionalize the automaton $\mathcal{A}(\hat{\Phi})$ with assignment η. Recalling that a first-order automaton has transitions labeled with first-order formulas, its propositionalization consists in first substituting the variables with values in η and then using function p to obtain propositional formulas in the transitions. We denote such an automaton with $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$;
2. We define a marking $M' := Q_0$ as the set of initial states and a marking $M = \emptyset$.
3. At runtime, when a new snapshot I is presented in input:
   (a) $M := M'$ and $M' := \emptyset$;
   (b) For each state $q \in M$ if there exists a transition $(q, \mathsf{p}(\gamma[\eta]), q')$ such that $\mathsf{p}(I) \models \mathsf{p}(\gamma[\eta])$, then $M' := M' \cup q'$.
   (c) We check the emptiness for all $q \in M'$. If there is at least one state that satisfies the Büchi condition, then we return ?, otherwise $false$.

The markings are needed to follow the execution of the snapshots over the automaton which is, in general, nondeterministic, hence more than one state can be contained in the current marking. If from none of the states in the marking it is possible to reach a cycle containing an accepting state, then the sequence of snapshots seen so far is a bad prefix for Φ, and the procedure returns $false$.

By running the same procedure in parallel on the automaton for $\neg\Phi$ (except for point 3 (c) where we output $true$ if none of the states satisfies the Büchi condition) we recognize also the good prefixes.

We show that this procedure capture exactly the semantics in Definition 7.3. For proving this claim, we reduce to the propositional case (where the same result has been proved to hold in [14]) by showing that the automaton $\mathsf{p}(\mathfrak{A}[\eta])$ and the

propositional one $\mathcal{A}[\eta]$ for $\mathsf{p}(\Phi[\eta])$ recognize the same language. If they do, by performing the emptiness check for each state, they recognize the same bad prefixes.

Theorem 7.1. *Given a formula $\Phi \in \mathcal{L}$, let $\hat{\Phi}$ be the (open) formula obtained from Φ by dropping the quantifiers and $\hat{\Phi}[\eta]$ the formula obtained from $\hat{\Phi}$ by substituting all variables with the value given by assignment η. Let moreover:*

- *$\mathcal{A}(\hat{\Phi})$ be the first order automaton for $\hat{\Phi}$;*
- *$\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta]))$ the propositional automaton for $\mathsf{p}(\hat{\Phi}[\eta])$;*
- *$\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ the automaton obtained from $\mathcal{A}(\hat{\Phi})$ by substituting variables with values given by assignment η and by propositionalizing first order formulas with function p;*

then $L(\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])) = L(\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta])))$.

Proof. We prove that the two languages are the same by showing that automata $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ and $\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta]))$ are the same automaton.

First of all, notice that p applied to formula $\hat{\Phi}$ that contains no quantifiers, is used for technical convenience only, given that, according to Lemma 7.2, it trivially associates atoms of the form $R(x, y)$ to propositional symbols R_{x_y}. We can hence abstract away p. As a result, we now have to prove that $\mathfrak{A}(\hat{\Phi})[\eta]$ and $\mathcal{A}(\hat{\Phi}[\eta])$ are the same automaton.

Assignment η can be viewed as a syntactic manipulation of the formula that changes the name of the variables. Two cases are possible: *(i)* η assigns each variable symbol to a different constant, and *(ii)* η assigns two (or more) variable symbols to the same constant. In the first case $\hat{\Phi}[\eta]$ is equal modulo atoms renaming to $\hat{\Phi}$, hence it is trivial to prove that $\mathcal{A}(\hat{\Phi})[\eta]$ is syntactically equal to $\mathcal{A}(\hat{\Phi}[\eta])$. The second case in more involved, since two different atoms, e.g., $R(x, y)$ and $R(x, x)$ can become identical, e.g., when $\eta = \{x/a, y/a\}$. In the rest of the proof, we assume that Φ contains two variables only, x and y, the extension to the general case in trivial. We write $\hat{\Phi}(x, y)$ to emphasize that formula $\hat{\Phi}$ has free variables x and y.

Putting all together, we hence have to prove that automata $\mathcal{A}(\hat{\Phi}(x, y))|_{y/x}$ and $\mathcal{A}(\hat{\Phi}(x, x))$ are indeed the same automaton, where with $\mathcal{A}(\hat{\Phi}(x, y))|_{y/x}$ we denote the automaton obtained from $\mathcal{A}(\hat{\Phi}(x, y))$ by syntactically replacing each occurrence of y with x. In this way we reduce to case *(i)* and we show the claim for each possible assignment η.

Let $A = \mathcal{A}(\hat{\Phi}(x, y))$ and $B = \mathcal{A}(\hat{\Phi}(x, x))$ Recalling Definition 7.5, the two automata are $A = (2^{Atom(\hat{\Phi})}, Q_A, \delta_A, Q_{A,0}, F_A)$ and $B = (2^{Atom(\hat{\Phi})}, Q_B, \delta_B, Q_{B,0}, F_B)$ where $Atom(\hat{\Phi})$ is the set of atoms of $\hat{\Phi}$.

We show that $Q_A|_{y/x} = Q_B$. By definition, Q_A is the set $\boldsymbol{B} = \{B_1 \dots B_n\}$ of all elementary formulas $B_i \subseteq cl(\hat{\Phi})$, and after the substitution y/x it can happen that some B_i is not consistent, hence they cannot appear in $Q_A|_{y/x}$. Let us assume, by contradiction, that there exists $\overline{B}_j \in Q_B$ such that $\overline{B} \notin Q_A|_{y/x}$. Substitution y/x can map different symbols into the same one, then if $\overline{B} \notin Q_A|_{y/x}$, this means that we obtained it as a result of the substitution but we dropped it because inconsistent. This is a contradiction given that $\overline{B}$ is in Q_B and hence it should be consistent. As a result, $Q_A|_{y/x} = Q_B$.

Given $Q_A|_{y/x} = Q_B$, it is trivial to prove that $Q_{A,0}|_{y/x} = Q_{0,B}$ and $F_A|_{y/b} = F_B$.

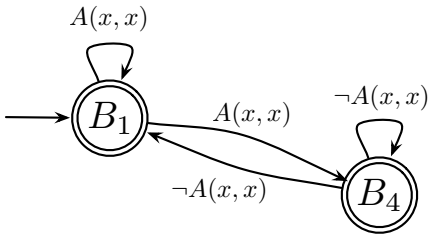


Figure 7.2. Graphical representation of the automaton $\mathcal{A}(\hat{\Phi}(x,y))$ in Example 7.2.2.

We show that $\delta_A|_{y/x} = \delta_B$. For each atom $a \in 2^{Atom(\hat{\Phi})}$, by definition, if $a \neq B \cap Atom(\hat{\Phi})$, then $\delta(B,a) = \emptyset$, then transition with atoms which include atoms with y as a variables are automatically ruled out. By Definition of δ, second bullet in 7.5, it is easy to see that transitions are defined starting from the set of elementary formulas, which we already shown to be equal, and from the set $cl(\hat{\Phi}(x,y))|_{y/x}$, which is trivially equal to $cl(\hat{\Phi}(x,x))$. We hence get $\delta_A|_{y/x} = \delta_B$. □

The proof is based on the observation that the two automata have a very similar structure. Indeed, the first-order automaton $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ can have more states and transitions than $\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta]))$, because different atoms in $\hat{\Phi}$, e.g., $R(x)$ and $R(y)$, are treated as different when constructing $\mathfrak{A}(\Phi)$, but they can be the same propositional symbol in $\mathsf{p}(\Phi[\eta])$, e.g., if $\eta = \{x/a, y/a\}$ (the vice-versa does not apply).

Moreover, by checking the emptiness per state at each step, we are guaranteed to find the *minimal* bad prefixes.

Example 7.2.2. *We show the automata $\mathcal{A}(\hat{\Phi}(x,x))$ for the formula $\Phi(x,x) = \forall x,y.A(x,x)\ \boldsymbol{U}\ A(x,x)$ using the usual procedure in [12]. We then illustrate how the same automaton is obtained by using $\mathcal{A}(\hat{\Phi}(x,y))$ and substitution $\{y/x\}$.*

Formula $\hat{\Phi} = A(x,x)\ \boldsymbol{U}\ A(x,x)$ and $cl(\hat{\Phi}) = \{A(x,x), \neg A(x,x), A(x,x)\ \boldsymbol{U}\ A(x,x), \neg(A(x,x)\ \boldsymbol{U}\ A(x,x))\}$. The set of elementary formulas is:

- $B_1 = \{A(x,x), A(x,x)\ \boldsymbol{U}\ A(x,x)\}$;
- $B_4 = \{\neg A(x,x), \neg(A(x,x)\ \boldsymbol{U}\ A(x,x))\}$.

A graphical representation of $\mathcal{A}(\hat{\Phi}(x,x))$ is in Figure 7.2.

Let us now analyze the automaton $\mathcal{A}(\hat{\Phi}(x,y))|_{y/x}$. The set of its elementary formulas is:

- $B_1|_{y/x} = \{A(x,x), A(x,x)\ \boldsymbol{U}\ A(x,x)\}$;
- $B_2|_{y/x} = \{\neg A(x,x), A(x,x), A(x,x)\ \boldsymbol{U}\ A(x,x)\}$ *(inconsistent);*
- $B_3|_{y/x} = \{A(x,x), \neg A(x,x), A(x,x)\ \boldsymbol{U}\ A(x,x)\}$ *(inconsistent);*
- $B_4|_{y/x} = \{\neg A(x,x), \neg(A(x,x)\ \boldsymbol{U}\ A(x,x))\}$;
- $B_5|_{y/x} = \{A(x,x), \neg A(x,x), \neg(A(x,x)\ \boldsymbol{U}\ A(x,x))\}$ *(inconsistent).*

By dropping inconsistent states and transitions from $\mathcal{A}(\hat{\Phi}(x,y))$, it is easy to see that $\mathcal{A}(\hat{\Phi}(x,y))|_{y/x}$ is the automaton $\mathcal{A}(\hat{\Phi}(x,x))$ represented in Figure 7.2.

We now show a key idea for monitoring the whole formula Φ, i.e., all assignment concurrently. Since Φ is in prenex normal form, formula $\mathsf{p}(\Phi)$ has hence the structure $\bigwedge_{d_1 \in \Delta} \bigvee_{d_2 \in \Delta} \cdots \bigwedge_{d_{n+m} \in \Delta} \mathsf{p}(\Phi[x/d_1, y/d_2 \ldots z/d_{n+m}])$. We can look for the bad (or good) prefixes of such a formula by monitoring $\mathsf{p}(\Phi[\eta])$ for each assignment η independently and then composing the results. Indeed, from Definition 7.3 and the semantics of LTL, it follows that:

- $[\pi[k] \models \phi_1 \wedge \phi_2] = true$ iff $[\pi[k] \models \phi_1] = true \vee [\pi[k] \models \phi_2] = true$;
- $[\pi[k] \models \phi_1 \wedge \phi_2] = false$ iff $[\pi[k] \models \phi_1] = false \wedge [\pi[k] \models \phi_2] = false$;
- $[\pi[k] \models \phi_1 \wedge \phi_2] =?$ otherwise

and analogously for $[\pi[k] \models \phi_1 \vee \phi_2]$.

Given a first order automaton $\mathcal{A}(\hat{\Phi})$, the procedure for monitoring Φ is as follows:

1. We define a marking $m : Q \to 2^{\vec{\eta}}$ (where $\vec{\eta}$ is the set of all possible assignment for Φ) as a function that takes in input a state q of $\mathcal{A}(\hat{\Phi})$ and return the set of assignments q is marked with; we assign $m'(q) = \vec{\eta}$ for each $q \in Q_0$ and $m'(q) = \emptyset$ for each $q \notin Q_0$.
2. When a new snapshot I is presented in input:
   (a) $m := m'$ and $m'(q) = \emptyset$ for each $q \in Q$.
   (b) for each state q and for each assignment $\eta \in m(q)$ if there exists a transition (q, γ, q') such that $I \models \gamma[\eta]$ (recall that γ is a first-order formula) then $m'(q') = m'(q') \cup \eta$.
   (c) for each assignment η we assign a truth value $t(\eta)$ as follows: if there exists at least one state q such that $\eta \in m'(q)$ and the emptiness check from q w.r.t. η (see later) returns true, then $t(\eta) =?$, otherwise $t(\eta) = false$.
   (d) we compose the truth values obtained according to the shape of Φ.

Notice that the emptiness check is now more complex, because (unlike the previous case) transitions are first-order formulas. In order to guarantee to find the minimal bad prefixes, for each q and η such that $\eta \in m'(q)$, we have to substitute the assignment η to all transitions involved in paths starting from q. We cannot just check if there exists a path that satisfies the Büchi condition from q, since there can be transitions (q, γ, q') that are first-order satisfiable, i.e., there is at least one assignment that satisfies γ, but actually $\gamma[\eta]$ is unsatisfiable. We have indeed to be sure that $\mathcal{A}(\hat{\Phi})$ is suitably pruned from unsatisfiable transitions for a given assignment η, before checking for emptiness for η.

By reducing to the previous case, it is easy to see that this procedure captures the semantics of Definition 7.3 for a first-order formula Φ and furthermore recognizes minimal good and bad prefixes, i.e., it reports a violation or satisfaction at the earliest possible.

7.3 Time and space complexity

The number of states of the automaton for the formula Φ is $c^{|\Phi|}$. During the runtime monitoring we keep $|\Delta|^n$ number of assignments where n is the number of variables,

hence, in the worst case $|\Delta|^{|\Phi|}$. Given that the automaton is nondeterministic, each state can be marked with all the possible assignments, we get a space complexity of $c^{|\Phi|} \cdot |\Delta|^{|\Phi|}$ which is still exponential in the size of the formula. Recall that the naive approach costs $c^{|\Delta|^{|\Phi|}}$ in space.

This gain does not come for free. Indeed, while in the naive approach the emptiness per state can be done offline and once for all after the construction of the automaton, in our case checking the Büchi condition is more involved, because we have to perform it separately for each tuple. As described in the previous Section, we perform such an analysis on the fly. Assuming we use nested depth first search, that is linear time in the size of the automaton, each time a new instance is presented as input, we have to check the emptiness $|\Delta|^{|\Phi|}$ times. We get $|\Delta|^{|\Phi|} \cdot c^{|\Phi|}$ time complexity at each step.

Notice that we could check the emptiness for each assignment offline, paying the time cost once for all. This, however, would require to keep for each state and for each assignment the information about the badness of that state for that specific tuple, leading to another $c^{|\Phi|} \cdot |\Delta|^{|\Phi|}$ in space. Since the major constraint of this problem is the space rather than the time, we prefer to perform the analysis online.

7.4 Discussion

In this chapter we have shown an automata-based mechanism that can be used to incrementally evaluate temporal queries over data evolutions that is is considerably less expensive than the naive monitoring of the propositionalized property.

This work stands in the middle-ground between databases and verification, hence it has been influenced by both such fields.

Concerning databases, works on incremental evaluation of queries are closely related to our approach. In [59] an incremental evaluation systems that uses first-order queries to incrementally evaluate recursive (i.e. fixpoint) queries is presented, while Chomicki [38] focuses on an "history less" approach for checking database integrity constraints using views. Also Toman [128] provides techniques for incremental query evaluation, but they are based on query specialization and physical deletion of past databases rather then views. All these solutions do not mention automata. Besides, how to evaluate a temporal queries over databases has been extensively studied in database literature and ad-hoc query languages has been proposed, such as TSQL2 [122]. Such approaches, however, explicitly represent time in the data, hence they deeply differ from the incremental approach used here.

Concerning runtime verification, different formalisms may be used for specifying admissible executions, such as ω-regular languages [48], LTL [14] or even μ-calculus [49], but all of them uses propositional languages.

In [13] open first-order temporal properties are monitored, and the technique proposed returns assignments that falsifies the formula. However, the logic is too expressive for supporting satisfiability and, more important, there is no "lookahead" mechanism of possible future evolutions (automata are not used indeed) so the bad prefixes recognized are not minimal. The work in [79] is the closest to ours, but no emphasis on the complexity nor optimizations is placed.

Bibliography

[1] Abdulla, P. A., Cerans, K., Jonsson, B., and Tsay, Y.-K. General decidability theorems for infinite-state systems. In *LICS*, pp. 313–321 (1996).

[2] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley (1995).

[3] Abiteboul, S., Vianu, V., Fordham, B. S., and Yesha, Y. Relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, **61** (2000), 236.

[4] Apt, K. R., Blair, H. A., and Walker, A. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pp. 89–148. Morgan Kaufmann (1988). ISBN 0-934613-40-0.

[5] Artale, A., Calvanese, D., Kontchakov, R., and Zakharyaschev, M. The *DL-Lite* family and relations. *J. of Artificial Intelligence Research*, **36** (2009), 1.

[6] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F. (eds.). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press (2003).

[7] Baader, F., Ghilardi, S., and Lutz, C. LTL over description logic axioms. *ACM Trans. on Computational Logic*, **13** (2012).

[8] Bagheri Hariri, B., Calvanese, D., De Giacomo, G., and De Masellis, R. Verification of conjunctive-query based semantic artifacts. In *Proc. of the 24th Int. Workshop on Description Logic (DL 2011)*, vol. 745 of *CEUR Electronic Workshop Proceedings,* http://ceur-ws.org/ (2011).

[9] Bagheri Hariri, B., Calvanese, D., De Giacomo, G., De Masellis, R., and Felli, P. Foundations of relational artifacts verification. In *Proc. of the 9th Int. Conference on Business Process Management (BPM 2011)*, vol. 6896 of *Lecture Notes in Computer Science*, pp. 379–395. Springer (2011).

[10] Bagheri Hariri, B., Calvanese, D., De Giacomo, G., De Masellis, R., Felli, P., and Montali, M. Description logic knowledge and action bases. *J. of Artificial Intelligence Research*, (2013).

[11] Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., and Montali, M. Verification of relational data-centric dynamic systems with external services. In *PODS*, pp. 163–174 (2013).

[12] Baier, C. and Katoen, J.-P. *Principles of model checking.* MIT Press (2008).

[13] Basin, D. A., Klaedtke, F., and Müller, S. Policy monitoring in first-order temporal logic. In *CAV*, pp. 1–18 (2010).

[14] Bauer, A., Leucker, M., and Schallhart, C. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, **20** (2011), 14:1.

[15] Beeri, C. and Vardi, M. Y. The implication problem for data dependencies. In *Proc. of ICALP'81*, vol. 115 of *Lecture Notes in Computer Science*, pp. 73–85. Springer (1981).

[16] Belardinelli, F., Lomuscio, A., and Patrizi, F. A computationally-grounded semantics for artifact-centric systems and abstraction results. In *IJCAI*, pp. 738–743 (2011).

[17] Belardinelli, F., Lomuscio, A., and Patrizi, F. Verification of deployed artifact systems via data abstraction. In *Proc. of the 9th Int. Joint Conf. on Service Oriented Computing (ICSOC 2011)*, vol. 7084 of *Lecture Notes in Computer Science*, pp. 142–156. Springer (2011).

[18] Belardinelli, F., Lomuscio, A., and Patrizi, F. An abstraction technique for the verification of artifact-centric systems. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2012)*, pp. 319–328 (2012).

[19] Belardinelli, F., Lomuscio, A., and Patrizi, F. Verification of gsm-based artifact-centric systems through finite abstraction. In *ICSOC*, pp. 17–31 (2012).

[20] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., and Schnoebelen, P. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer Publishing Company, Incorporated, 1st edn. (2010). ISBN 3642074782, 9783642074783.

[21] Berardi, D., Calvanese, D., and De Giacomo, G. Reasoning on UML class diagrams. *Artificial Intelligence*, **168** (2005), 70.

[22] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Mecella, M. Automatic composition of e-services that export their behavior. In *ICSOC*, pp. 43–58 (2003).

[23] Bhattacharya, K., Gerede, C., Hull, R., Liu, R., and Su, J. Towards formal analysis of artifact-centric business process models. In *Proc. of the 5th Int. Conference on Business Process Management (BPM 2007)*, vol. 4714 of *Lecture Notes in Computer Science*, pp. 288–234. Springer (2007).

[24] Bouajjani, A., Habermehl, P., and Vojnar, T. Abstract regular model checking. In *CAV*, pp. 372–386 (2004).

[25] Bouajjani, A., Jonsson, B., Nilsson, M., and Touili, T. Regular model checking. In *CAV*, pp. 403–418 (2000).

[26] Brachman, R. J. and Levesque, H. J. *Knowledge Representation and Reasoning*. Elsevier (2004). ISBN 978-1-55860-932-7.

[27] Bradfield, J. and Stirling, C. Modal mu-calculi. In *Handbook of Modal Logic*, vol. 3, pp. 721–756. Elsevier (2007).

[28] Bryant, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, **24** (1992), 293.

[29] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodríguez-Muro, M., and Rosati, R. Ontologies and databases: The *DL-Lite* approach. In *Semantic Technologies for Informations Systems – 5th Int. Reasoning Web Summer School (RW 2009)* (edited by S. Tessaris and E. Franconi), vol. 5689 of *Lecture Notes in Computer Science*, pp. 255–356. Springer (2009).

[30] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. EQL-Lite: Effective first-order query processing in description logics. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007)*, pp. 274–279 (2007).

[31] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning*, **39** (2007), 385.

[32] Calvanese, D., De Giacomo, G., Lembo, D., Montali, M., and Santoso, A. Ontology-based governance of data-aware processes. In *Proc. of the 6th Int. Conf. on Web Reasoning and Rule Systems (RR 2012)*, vol. 7497 of *Lecture Notes in Computer Science*, pp. 25–41. Springer (2012).

[33] Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Actions and programs over description logic ontologies. In *Proc. of the 20th Int. Workshop on Description Logic (DL 2007)*, vol. 250 of *CEUR Electronic Workshop Proceedings,* http://ceur-ws.org/, pp. 29–40 (2007).

[34] Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Actions and programs over description logic knowledge bases: A functional approach. In *Knowing, Reasoning, and Acting: Essays in Honour of Hector Levesque* (edited by G. Lakemeyer and S. A. McIlraith). College Publications (2011).

[35] Calvanese, D., Giacomo, G. D., Hull, R., and Su, J. Artifact-centric workflow dominance. In *ICSOC/ServiceWave*, pp. 130–143 (2009).

[36] Cangialosi, P., De Giacomo, G., De Masellis, R., and Rosati, R. Conjunctive artifact-centric services. In *Proc. of the 8th Int. Joint Conf. on Service Oriented Computing (ICSOC 2010)*, vol. 6470 of *Lecture Notes in Computer Science*, pp. 318–333. Springer (2010).

[37] Chandra, A. K. and Merlin, P. M. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pp. 77–90 (1977).

[38] Chomicki, J. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, **20** (1995), 149.

[39] Clark, K. L. Negation as failure. In *Logic and Data Bases*, pp. 293–322 (1977).

[40] Clarke, E. M. and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pp. 52–71 (1981).

[41] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, **50** (2003), 752.

[42] Clarke, E. M., Grumberg, O., and Long, D. E. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, **16** (1994), 1512.

[43] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model checking*. The MIT Press, Cambridge, MA, USA (1999).

[44] Cohn, D. and Hull, R. Business artifacts: A data-centric approach to modeling business operations and processes. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, **32** (2009), 3.

[45] Courcelle, B. Monadic second-order definable graph transductions: A survey. *Theor. Comput. Sci.*, **126** (1994), 53.

[46] Damaggio, E., Deutsch, A., and Vianu, V. Artifact systems with data dependencies and arithmetic. In *Proc. of the 14th Int. Conf. on Database Theory (ICDT 2011)*, pp. 66–77 (2011).

[47] Damaggio, E., Hull, R., and Vaculín, R. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. In *BPM*, pp. 396–412 (2011).

[48] d'Amorim, M. and Rosu, G. Efficient monitoring of omega-languages. In *CAV*, pp. 364–378 (2005).

[49] D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S., and Manna, Z. Lola: Runtime monitoring of synchronous systems. In *TIME*, pp. 166–174 (2005).

[50] De Giacomo, G., De Masellis, R., and Patrizi, F. Composition of partially observable services exporting their behaviour. In *ICAPS* (2009).

[51] De Giacomo, G., De Masellis, R., and Rosati, R. Verification of conjunctive artifact-centric services. *Int. J. of Cooperative Information Systems*, **21** (2012), 111.

[52] De Giacomo, G., Iocchi, L., Nardi, D., and Rosati, R. A theory and implementation of cognitive mobile robots. *J. of Logic and Computation*, **9** (1999), 759.

[53] De Giacomo, G., Lespérance, Y., and Patrizi, F. Bounded situation calculus action theories and decidable verification. In *KR* (2012).

[54] De Giacomo, G., Lespérance, Y., and Patrizi, F. Bounded epistemic situation calculus theories. In *IJCAI* (2013).

[55] De Masellis, R., Di Ciccio, C., Mecella, M., and Patrizi, F. Smart home planning programs. In *ICSSSM*, pp. 377–382 (2010).

[56] De Masellis, R. and Su, J. Runtime enforcement of first-order ltl properties on data-aware business processes. In *ICSOC* (2013). To appear.

[57] Deutsch, A., Hull, R., Patrizi, F., and Vianu, V. Automatic verification of data-centric business processes. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT 2009)*, pp. 252–267 (2009).

[58] Deutsch, A., Nash, A., and Remmel, J. B. The chase revisited. In *PODS*, pp. 149–158 (2008).

[59] Dong, G., Su, J., and Topor, R. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, **14** (1995), 187.

[60] Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., and Campenhout, D. V. Reasoning with temporal logic on truncated paths. In *CAV*, pp. 27–39 (2003).

[61] Emerson, E. A. Automated temporal reasoning about reactive systems. In *Logics for Concurrency: Structure versus Automata* (edited by F. Moller and G. Birtwistle), vol. 1043 of *Lecture Notes in Computer Science*, pp. 41–101. Springer (1996).

[62] Emerson, E. A. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, pp. 185–214 (1996).

[63] Emerson, E. A. and Sistla, A. P. Deciding full branching time logic. *Information and Control*, **61** (1984), 175.

[64] Esparza, J. Decidability and complexity of petri net problems - an introduction. In *Petri Nets*, pp. 374–428 (1996).

[65] Fagin, R., Kolaitis, P. G., Miller, R. J., and Popa, L. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, **336** (2005), 89.

[66] Finkel, A. and Schnoebelen, P. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, **256** (2001), 63.

[67] Fischer, M. J. and Ladner, R. E. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, **18** (1979), 194.

[68] Fitting, M. and Mendelsohn, R. L. *First-Order Modal Logic*. Kluwer Academic Press (1998).

[69] Fritz, C., Hull, R., and Su, J. Automatic construction of simple artifact-based business processes. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT 2009)*, pp. 225–238 (2009).

[70] Gabbay, D., Kurusz, A., Wolter, F., and Zakharyaschev, M. *Many-dimensional Modal Logics: Theory and Applications*. Elsevier Science Publishers (2003).

[71] Gelder, A. V. Negation as failure using tight derivations for general logic programs. In *SLP*, pp. 127–138 (1986).

[72] Gerede, C. E., Bhattacharya, K., and Su, J. Static analysis of business artifact-centric operational models. In *SOCA*, pp. 133–140 (2007).

[73] Gerede, C. E. and Su, J. Specification and verification of artifact behaviors in business process models. In *ICSOC*, pp. 181–192 (2007).

[74] Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, pp. 3–18 (1995).

[75] Giacomo, G. D. and Sardiña, S. Automatic synthesis of new behaviors from a library of available behaviors. In *IJCAI*, pp. 1866–1871 (2007).

[76] Gonzalez, P., Griesmayer, A., and Lomuscio, A. Verifying gsm-based business artifacts. In *ICWS*, pp. 25–32 (2012).

[77] Grefen, P. W. P. J. and de By, R. A. A multi-set extended relational algebra - a formal approach to a practical issue. In *ICDE*, pp. 80–88 (1994).

[78] Gu, Y. and Soutchanski, M. A description logic based situation calculus. *Ann. of Mathematics and Artificial Intelligence*, **58** (2010), 3.

[79] Hallé, S. and Villemaire, R. Runtime monitoring of message-based workflows with data. In *EDOC*, pp. 63–72 (2008).

[80] Hariri, B. B., Calvanese, D., De Giacomo, G., De Masellis, R., Felli, P., and Montali, M. Verification of description logic knowledge and action bases. In *ECAI*, pp. 103–108 (2012).

[81] Hull, R. Artifact-centric business process models: Brief survey of research results and challenges. In *Proc. of ODBASE 2008*, vol. 5332 of *Lecture Notes in Computer Science*, pp. 1152–1163. Springer (2008).

[82] Hull, R., et al. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *WS-FM*, pp. 1–24 (2010).

[83] Hull, R., et al. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *DEBS*, pp. 51–62 (2011).

[84] Jensen, K. and Kristensen, L. M. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer (2009). ISBN 978-3-642-00283-0.

[85] Jonsson, B. and Nilsson, M. Transitive closures of regular relations for verifying infinite-state systems. In *TACAS*, pp. 220–234 (2000).

[86] Katsuno, H. and Mendelzon, A. On the difference between updating a knowledge base and revising it. In *Proc. of the 2nd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'91)*, pp. 387–394 (1991).

[87] Kolaitis, P. G. Schema mappings, data exchange, and metadata management. In *Proc. of PODS 2005*, pp. 61–75 (2005).

[88] Künzle, V. and Reichert, M. PHILharmonicFlows: towards a framework for object-aware process management. *Journal of Software Maintenance and Evolution: Research and Practice*, **23** (2011), 205.

[89] Künzle, V., Weber, B., and Reichert, M. Object-aware business processes: Fundamental requirements and their support in existing approaches. *IJISMD*, **2** (2011), 19.

[90] Kupferman, O. and Vardi, M. Y. Model checking of safety properties. *Formal Methods in System Design*, **19** (2001), 291.

[91] Lamperti, G., Melchiori, M., and Zanella, M. On multisets in database systems. In *WMP*, pp. 147–216 (2000).

[92] Lenzerini, M. Data integration: A theoretical perspective. In *PODS*, pp. 233–246 (2002).

[93] Levesque, H. J. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, **23** (1984), 155.

[94] Limonad, L., De Leenheer, P., Linehan, M., Hull, R., and Vaculin, R. Ontology of dynamic entities. In *Proc. of the 31st Int. Conf. on Conceptual Modeling (ER 2012)* (2012).

[95] Lin, F. and Reiter, R. State constraints revisited. *J. of Logic Programming*, **4** (1994), 655.

[96] Linehan, M. GSM expression language. Tech. rep., IBM Research (2011). Available on request.

[97] Liu, H., Lutz, C., Milicic, M., and Wolter, F. Reasoning about actions using description logics with general TBoxes. In *Proc. of the 10th Eur. Conference on Logics in Artificial Intelligence (JELIA 2006)*, vol. 4160 of *Lecture Notes in Computer Science*. Springer (2006).

[98] Liu, H., Lutz, C., Milicic, M., and Wolter, F. Updating description logic ABoxes. In *Proc. of the 10th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2006)*, pp. 46–56 (2006).

[99] Lloyd, J. W. *Foundations of Logic Programming, 2nd Edition*. Springer (1987). ISBN 3-540-18199-7.

[100] Luckham, D. C., Park, D. M. R., and Paterson, M. On formalised computer programs. *J. of Computer and System Sciences*, **4** (1970), 220.

[101] Lutz, C., Wolter, F., and Zakharyaschev, M. Temporal description logics: A survey. In *Proc. of the 15th Int. Symp. on Temporal Representation and Reasoning (TIME 2008)*, pp. 3–14 (2008).

[102] Manna, Z. and Pnueli, A. *Temporal verification of reactive systems - safety*. Springer (1995). ISBN 978-0-387-94459-3.

[103] Marnette, B. Generalized schema-mappings: from termination to tractability. In *PODS*, pp. 13–22 (2009).

[104] Marnette, B. and Geerts, F. Static analysis of schema-mappings ensuring oblivious termination. In *ICDT*, pp. 183–195 (2010).

[105] McCarthy, J. Situations, actions and causal laws. Tech. rep., Stanford University (1963). Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[106] Meier, M., Schmidt, M., and Lausen, G. On chase termination beyond stratification. *PVLDB*, **2** (2009), 970.

[107] Meyer, A., Smirnov, S., and Weske, M. Data in business processes. *EMISA Forum*, **31** (2011), 5.

[108] Milner, R. An algebraic definition of simulation between programs. In *IJCAI*, pp. 481–489 (1971).

[109] Nigam, A. and Caswell, N. S. Business artifacts: An approach to operational specification. *IBM Systems Journal*, **42** (2003), 428.

[110] OASIS. Web Services Business Process Execution Language (WS-BPEL) ver. 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (2007).

[111] Object Management Group (OMG). Business Process Model and Notation (BPMN) ver. 2.0. http://www.omg.org/spec/BPMN/2.0 (2011).

[112] Object Management Group (OMG). Case Management Model And Notation vers. 1.0 - Beta 1. http://www.omg.org/spec/CMMN/1.0/Beta1/PDF/ (2013).

[113] Park, D. M. R. Finiteness is mu-ineffable. *Theor. Comput. Sci.*, **3** (1976), 173.

[114] Patrizi, F. *Simulation-Based Techniques for Automated Service Composition*. Ph.D. thesis, Sapienza Università degli Studi di Roma (2009).

[115] Pnueli, A. The temporal logic of programs. In *FOCS*, pp. 46–57 (1977).

[116] Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Linking data to ontologies. *J. on Data Semantics*, **X** (2008), 133.

[117] Reichert, M. and Weber, B. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer (2012). ISBN 978-3-642-30408-8.

[118] Reiter, R. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press (2001).

[119] Robinson, R. Undecidability and nonperiodicity of tilings on the plane. *Inventiones Math.*, **12** (1971), 177.

[120] Rosati, R. and Franconi, E. Generalized ontology-based production systems. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2012)*, pp. 435–445. AAAI Press/The MIT Press (2012).

[121] Schild, K. Combining terminological logics with tense logic. In *Proc. of the 6th Portuguese Conf. on Artificial Intelligence (EPIA'93)*, vol. 727 of *Lecture Notes in Computer Science*, pp. 105–120. Springer (1993).

[122] Snodgrass, R. T. (ed.). *The TSQL2 Temporal Query Language*. Kluwer (1995). ISBN 0-7923-9614-6.

[123] Sohrabi, S., Prokoshyna, N., and McIlraith, S. A. Web service composition via generic procedures and customizing user preferences. In *International Semantic Web Conference*, pp. 597–611 (2006).

[124] Solomakhin, D., Montali, M., Tessaris, S., and De Masellis, R. Verification of artifact-centric systems: Decidability and modeling issues. In *ICSOC* (2013). To appear.

[125] Stirling, C. *Modal and temporal properties of processes*. Springer-Verlag New York, Inc., New York, NY, USA (2001). ISBN 0-387-98717-7.

[126] Tarski, A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5** (1955), 285.

[127] ter Hofstede, A. H. M., van der Aalst, W. M. P., Adams, M., and Russell, N. (eds.). *Modern Business Process Automation - YAWL and its Support Environment*. Springer (2010). ISBN 978-3-642-03120-5.

[128] Toman, D. Expiration of historical databases. In *TIME*, pp. 128–135 (2001).

[129] Van Der Aalst, W. and Stahl, C. *Modeling Business Processes: A Petri Net-Oriented Approach*. Cooperative Information Systems Series. Mit Press (2011). ISBN 9780262015387.

[130] van der Aalst, W. M. P. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, **8** (1998), 21.

[131] van der Aalst, W. M. P. Formalization and verification of Event-driven Process Chains. *Information and Software Technology*, **41** (1999), 639.

[132] van der Aalst, W. M. P., Barthelmess, P., Ellis, C. A., and Wainer, J. Proclets: A framework for lightweight interacting workflow processes. *Int. J. of Cooperative Information Systems*, **10** (2001), 443.

[133] van der Aalst, W. M. P., ter Hofstede, A. H. M., and Weske, M. Business process management: A survey. In *Business Process Management*, pp. 1–12 (2003).

[134] van der Aalst, W. M. P. and van Hee, K. M. *Workflow Management: Models, Methods, and Systems*. MIT Press (2002). ISBN 0-262-01189-1.

[135] van Emde Boas, P. The convenience of tilings. In *Complexity, Logic, and Recursion Theory* (edited by A. Sorbi), vol. 187 of *Lecture Notes in Pure and Applied Mathematics*, pp. 331–363. Marcel Dekker Inc. (1997).

[136] Vardi, M. Y. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pp. 238–266 (1995).

[137] Walukiewicz, I. Monadic second-order logic on tree-like structures. *Theor. Comput. Sci.*, **275** (2002), 311.

[138] Weske, M. *Business Process Management: Concepts, Languages, Architectures*. Springer (2007).

[139] Wolter, F. and Zakharyaschev, M. Temporalizing description logic. In *Frontiers of Combining Systems* (edited by D. Gabbay and M. de Rijke), pp. 379–402. Studies Press/Wiley (1999).