UNIVERSITY OF CALIFORNIA, SAN DIEGO

Verification of Business Process Specifications With Arithmetic and Data Dependencies

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Elio Damaggio

Committee in charge:

Alin Deutsch, Co-Chair
Victor Vianu, Co-Chair
Richard Hull
Bertram Ludaescher
Yannis Papakonstantinou
Jeffrey Remmel

2011

The dissertation of Elio Damaggio is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Co-Chair

_____
Co-Chair

University of California, San Diego

2011

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Thanks to my advisors Alin Deutsch and Victor Vianu for guiding me and inspiring me during my work on this thesis. Special thanks to Rick Hull for making me part of the effort on the ArtiFact model, and to all the extended team ArtiFact at IBM Research. Finally, I want to thank the other co-authors of my papers: Dayou Zhou and Roman Vaculin.

Also, I explicitly acknowledge Alin Deutsch and Victor Vianu that co-authored Chapters 1, 2, 3 and 8 of this thesis; and Richard Hull who co-authored Chapter 1.

VITA

2003	Laurea in Dottore in Ingegneria Informatica *cum laude*, Università 'La Sapienza' di Roma, Italy
2003-2006	Senior Designer/Analyst at Value Team S.p.a., Italy
2007	Software Engineering Intern, Qualcomm, San Diego, CA
2008	Master of Science in Computer Science, University of California, San Diego
2008	Research Intern at Microsoft Research, Redmond, WA
2009	Research Intern at IBM Watson Research Center, Hawthorne, NY
2010	Research Intern at IBM Watson Research Center, Hawthorne, NY
2011	Doctor of Philosophy in Computer Science, University of California, San Diego

PUBLICATIONS

Damaggio, Hull, Vaculin, "On the equivalence of incremental and fixpoint semantics for business entities with guard-stage-milestone lifecycles", *International Conference on Business Process Management (BPM), 2011*.

Damaggio, Deutsch, Hull, Vianu, "Automatic Verification of data-Centric Business Process", *International Conference on Business Process Management (BPM), 2011*.

Damaggio, Deutsch, Zhou, "Querying contract databases based on temporal behavior", *International Conference on Management of Data (SIGMOD) 2011*.

Damaggio, Deutsch, Vianu, "Artifact Systems with Data Dependencies and Arithmetic", *International Conference on Database Theory (ICDT) 2011*.

Hull, Damaggio, De Masellis, Fournier, Gupta, Heath, Hobson, Linehan, Maradugu, Nigam, Sukaviriya, Vaculin, "Business entities with guard-stage-milestone lifecycles: Managing entity interactions with conditions and events", *Distributed Event-Based Systems (DEBS) 2011*.

Hull, Damaggio, Fournier, Gupta, Heath, Hobson, Linehan, Maradugu, Nigam, Sukaviriya, Vaculin, "Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles", *Web Services and Formal Models (WSFM) 2010*

Calvanese, Damaggio, De Giacomo, Lenzerini and Rosati, "Semantic Data Integration in Peer-to-Peer Architectures", *International Workshop on Databases, Information Systems and Peer-to-Peer Computing, 2003*.

ABSTRACT OF THE DISSERTATION

Verification of Business Process Specifications With Arithmetic and Data Dependencies

by

Elio Damaggio

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Alin Deutsch, Co-Chair
Victor Vianu, Co-Chair

Recent years have witnessed the evolution of business process specification frameworks from the traditional process-centric approach towards data-awareness. Process-centric formalisms focus on control flow while under-specifying the underlying data and its manipulations by the process tasks, often abstracting them away completely. In contrast, data-aware formalisms treat data as first-class citizens. The presence of data implies an increase expressiveness of business process specification, including often data dependencies and arithmetic. This thesis studies the verification problem of temporal properties on data-aware business specifications with data dependencies and arithmetic.

In our context, data implies infinite-state systems, for which verification is noto-

riously difficult. Unlike previous work, we focus on verification that is a) automatic (i.e. no expert user is required to help the process of verification as for theorem provers), b) sound and complete, and c) does not abstract away the data portion, retaining the ability to check the effects of data values on the behavior of the process (e.g. in a prototypical e-commerce business process, abstracting the data would make it impossible to check if the payment received for a product matches the price reported on the bill).

We identify a practically significant class of business process specifications with data dependencies and arithmetic, for which verification of temporal properties is decidable. Besides decidability, in the context of commonly occurring classes of specifications, we develop verification techniques with upper bounds palatable to implementation, e.g. PSPACE for a common class of specifications with unary keys and fixed-arity databases with acyclic foreign keys.

We implement a verifier prototype based on our theoretical results and measure the running times of the verification of temporal properties on a wide range of business process specifications of different complexities. Our random generation is based on patterns and frequencies found in real-world business process specifications and properties. The average running times measured range from seconds to minutes for the more complex specifications.

We argue that the work in this thesis proves the feasibility of automatic verification of temporal properties on highly expressive business process specifications, that is both sound and complete.

1 Introduction

Recent years have witnessed the evolution of business process specification frameworks from the traditional process-centric approach towards data-awareness. Process-centric formalisms focus on control flow while under-specifying the underlying data and its manipulations by the process tasks, often abstracting them away completely. In contrast, data-aware formalisms treat data as first-class citizens. A notable exponent of this class is the *business artifact model* pioneered in [NC03], deployed by IBM in commercial products and consulting services, and further studied in a line of follow-up works [BCK$^+$07, B$^+$05, GBS07, GS07, BGH$^+$07, LBW07, KLW08, KRG07, ea10]. Business artifacts (or simply "artifacts") model key business-relevant entities, which are updated by a set of services that implement business process tasks. A collection of artifacts and services is called an *artifact system*. This modeling approach has been successfully deployed in practice, yielding proven savings when performing business process transformations [BCK$^+$07]. In this thesis, we focus on automatic verification for data-aware business process specifications at the theoretical and implementation level.

The verification problem [VW86] consists, given a model of a business process, in establishing the validity of a temporal property, usually expressed in some flavor of temporal logic. The verification problem not only arises naturally in applications, for instance rule compliance (e.g. checking if a business process adheres to some company or government standard), but it is a fundamental building block for many other useful applications, as reported in Section 1.1. Traditional software verification techniques can be applied to data-aware business process models; however, they suffer from one of the following limitations. Theorem provers provide enough expressivity to encode data-aware business process models, however, they might require an expert user to help during the verification procedure. This kind of verification is sometimes called 'inter-

active'. In this thesis we aim to provide an 'automatic' verification technique that does not require the presence of an expert user to prove intermediate lemmas, whose connection to the initial problem is often very hard to grasp. Another approach, based on model checking techniques [VW86], is to abstract away the complexity deriving from the presence of the data. This results in the lack of the ability to check the effects of data values on the behavior of the process. For instance, in a prototypical e-commerce business process, it would be impossible to check if the payment received for a product matches the price reported on the bill. The approach we pursue in this thesis is then to provide an automatic technique for data-aware verification of temporal properties.

In [DHPV09], data-aware verification for business processes is introduced. These techniques, however, require conditions on specifications that are rarely satisfied in real world applications, namely no arithmetic operations and no integrity constraints on the external database. This would exclude any business process that computes an order value summing product's price and shipping cost, or refers to a database where the table storing product information has a key constraint. Reasoning with arithmetic operations and integrity constraints, even in isolation, leads quickly to undecidability. In Chapter 2 of this thesis, we present a syntactic way to identify a class of business process models with arithmetic operations and integrity constraints for which verification is feasible [DDV11]. Supported by surveys [VDATHKB03] and [AM00], we also believe that this restriction is expressive enough to cover a wide variety (if not most) of useful business process specifications.

In addition to this theoretical result, we performed a feasibility study on our verification technique. This study includes a higher level model for business process models that naturally relates to our syntactic restriction, and that form the basis of our experiments; the implementation of a prototype verifier based on the theoretical results that we presented; and a series of experiments aimed at validating the real-world performance of data-aware verification. In the rest of this chapter, we report other applications of verification and an extended summary of the main results of the thesis.

1.1 Other applications of verification

Various useful static analysis problems on data-aware business processes can be reduced to verification of temporal properties. We mention some of these.

1.1.1 Business rules

The basic artifact model is extended in [BGH+07] with *business rules*, in order to support service reuse and customization. Business rules are conditions that can be super-imposed on the pre-conditions of existing services without changing their implementation. They are useful in practice when services are provided by autonomous third-parties, who typically strive for wide applicability and impose as unrestrictive pre-conditions as possible. When such third-party services are incorporated into a specific business process, this often requires more control over when services apply, in the form of more restrictive pre-conditions. Such additional control may also be needed to ensure compliance with business regulations formulated by third parties, independently of the specific application. Verification of properties in the presence of business rules then becomes of interest and can be addressed by our techniques. A related issue is the detection of *redundant* business rules, which can also be reduced to a verification problem.

1.1.2 Redundant attributes

Another design simplification consists of redundant attribute removal, a problem also raised in [BGH+07]. This is formulated as follows. We would like to test whether there is a way to satisfy a property φ of runs without using one of the attributes. This easily reduces to a verification problem as well.

1.1.3 Verifying termination properties

In some applications, one would like to verify properties relating to termination, e.g. reachability of configurations in which no service can be applied. Note that one can

state, within an LTL-FO property, that a configuration of an artifact system is blocking. This allows reducing termination questions to verification.

1.2 Artifact Systems

In our work, data-aware business processes are specified following the business artifact paradigm [ea10]. For ease of presentation, we describe in this section a minimalistic variant of the artifact systems model fully described in Chapter 2. This variant is adequate for illustrating our approach to verification in this Chapter. The presentation is informal, relying mainly on a running example. The example, modeling an e-commerce process, features several characteristics that drive the motivation for our work.

1. The system routinely queries an underlying database, for instance to look up the price of a product and the shipping weight restrictions.

2. The validity checks and updates carried out by the services involve arithmetic operations. For instance, to be valid, an order must satisfy such conditions as: (a) the product weight must be within the selected shipment method's limit, and (b) if the buyer uses a coupon, the sum of product price and shipping cost must exceed the coupon's minimum purchase limit.

3. Finally, the correctness of the business process relies on database integrity constraints. For instance, the system must check that a selected triple of product, shipment type and coupon are globally compatible. This check is implemented by several local tests, each running at a distinct instant of the interaction, as user selections become available. Each local test accesses distinct tables in the database, yet they globally refer to the same product, due to the keys and foreign keys satisfied by these tables.

The example models an e-commerce business process in which the customer chooses a product and a shipment method and applies various kinds of coupons to the order. There are two kinds of coupons: discount coupons subtract their value from the total (e.g. a $50 coupon) and free-shipment coupons subtract the shipping costs from the total. The order is filled in a sequential manner (first pick the product, then the shipment, then claim a coupon), as is customary on e-commerce web-sites. After the order is filled, the system awaits for the customer to submit a payment. If the payment

matches the amount owed, the system proceeds to shipping the product.

As mentioned earlier, an artifact is an evolving record of values. The values are referred to by variables (sometimes called *attributes*). In general, an artifact system consists of several artifacts, evolving under the action of *services*, specified by pre- and post-conditions. In the example, we use a single artifact with the following variables

`status`,`prod_id`,`ship_type`,`coupon`,`amount_owed`,`amount_paid`,`amount_refunded`.

The `status` variable tracks the status of the order and can take the following values:

"edit_product", "edit_ship", "edit_coupon", "processing", "received_payment", "shipping", "shipped", "canceling", "canceled".

Artifact variables `ship_type` and `coupon` record the customer's selection, received as an external input. `amount_paid` is also an external input (from the customer, possibly indirectly via a credit card service). Variable `amount_owed` is set by the system using arithmetic operations that sum up product price and shipment cost, subtracting the coupon value. Variable `amount_refunded` is set by the system in case a refund is activated.

The database includes the following tables, where underlined attributes denote keys. Recall that a key is an attribute that uniquely identifies each tuple in a relation.

$$\texttt{PRODUCTS}(\underline{\texttt{id}}, \texttt{price}, \texttt{availability}, \texttt{weight}),$$
$$\texttt{COUPONS}(\underline{\texttt{code}}, \texttt{type}, \texttt{value}, \texttt{min\_value}, \texttt{free\_shiptype}),$$
$$\texttt{SHIPPING}(\underline{\texttt{type}}, \texttt{cost}, \texttt{max\_weight}),$$
$$\texttt{OFFERS}(\underline{\texttt{prod\_id}}, \texttt{discounted\_price}, \texttt{active}).$$

The database also satisfies the following foreign keys:

$$\texttt{COUPONS}[\texttt{free\_shiptype}] \subseteq \texttt{SHIPPING}[\texttt{type}] \text{ and}$$
$$\texttt{OFFERS}[\texttt{prod\_id}] \subseteq \texttt{PRODUCTS}[\texttt{id}].$$

Thus, the first dependency says that each free_shiptype value in the COUPONS relation is also a type value in the SHIPPING relation. The second inclusion dependency states that every prod_id value in the OFFERS is the actual ⊐ of a product in the PRODUCTS relation

The starting configuration of every artifact system is constrained by an initialization condition, which here states that status initialized to "edit_prod", and all other variables to "undefined". By convention, we model undefined variables using the reserved constant λ.

1.2.1 Services

Recall that artifacts evolve under the action of services. Each service is specified by a pre-condition π and a postcondition ψ, both existential first-order ($\exists$FO) sentences. The pre-condition refers to the current values of the artifact variables and the database. The post-condition ψ refers simultaneously to the current and *next* artifact values, as well as the database. In addition, both π and ψ may use arithmetic constraints on the variables, limited to linear inequalities over the rationals.

The following services model a few of the business process tasks of the example. Throughout the example, we use primed artifact variables x' to refer to the *next* value of variable x.

choose_product: The customer chooses a product.

π : status = "edit_prod"

$\psi : \exists p,a,w(\texttt{PRODUCTS}(\texttt{prod\_id}',p,a,w) \wedge a > 0) \wedge \texttt{status}' =$ "edit_shiptype"

choose_shiptype: The customer chooses a shipping option.

π : status = "edit_ship"

$\psi : \exists c,l,p,a,w(\texttt{SHIPPING}(\texttt{ship\_type}',c,l) \wedge \texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge l > w) \wedge$
status$'$ = "edit_coupon" $\wedge$ prod_id$'$ = prod_id

apply_coupon: The customer optionally inputs a coupon number.

π : status = "edit_coupon"

$\psi : (\texttt{coupon}' = \lambda \wedge \exists p,a,w,c,l(\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge$
$\texttt{SHIPPING}(\texttt{ship\_type},c,l) \wedge \texttt{amount\_owed}' = p + c) \wedge \texttt{status}' =$ "processing"

$\wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type}) \vee$
$(\exists t, v, m, s, p, a, w, c, l(\texttt{COUPONS}(\texttt{coupon}', t, v, m, s) \wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge \texttt{SHIPPING}(\texttt{ship\_type}, c, l) \wedge p + c \geq m \wedge$
$(t = \text{“free\_shipping”} \rightarrow (s = \texttt{ship\_type} \wedge \texttt{amount\_owed}' = p)) \wedge$
$(t = \text{“discount”} \rightarrow \texttt{amount\_owed}' = p + c - v))$
$\wedge \texttt{status}' = \text{“processing”} \wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type})$

Notice that the pre-conditions of the services check the value of the `status` variable. For instance, according to **choose_product**, the customer can only input her product choice while the order is in "edit_prod" status.

Also notice that the post-conditions constrain the next values of the artifact variables (denoted by a prime). For instance, according to **choose_product**, once a product has been picked, the next value of the status variable is "edit_shiptype", which will at a subsequent step enable the **choose_shiptype** service (by satisfying its pre-condition). Similarly, once the shipment type is chosen (as modeled by service **choose_shiptype**), the new status is "edit_coupon", which enables the **apply_coupon** service. The interplay of pre- and post-conditions achieves a sequential filling of the order, starting from the choice of product and ending with the claim of a coupon.

A post-condition may refer to both the current and next values of the artifact variables. For instance, in service **choose_shiptype**, the fact that only the shipment type is picked while the product remains unchanged, is modeled by preserving the product id: the next and current values of the corresponding artifact variable are set equal.

Pre- and post-conditions may query the database. For instance, in service **choose_product**, the post-condition ensures that the product id chosen by the customer is that of an available product (by checking that it appears in a `PRODUCTS` tuple, whose availability attribute is positive).

Finally, notice the arithmetic computation in the post-conditions. For instance, in service **apply_coupon**, the sum of the product price p and shipment cost c (looked up in the database) is adjusted with the coupon value (notice the distinct treatment of the two coupon types) and stored in the `amount_owed` artifact variable.

Observe that the first post-condition disjunct models the case when the customer inputs no coupon number (the next value $\texttt{coupon}'$ is set to undefined), in which case a

different owed amount is computed, namely the sum of price and shipping cost.

1.2.2 Semantics

The semantics of an artifact system $\mathscr{A}$ consists of its *runs*. Given a database D, a run of $\mathscr{A}$ is an infinite sequence $\{\rho_i\}_{\geq 0}$ of artifact records such that ρ_0 and D satisfy the initial condition of the system, and for each $i \geq 0$ there is a service S of the system such that ρ_i and D satisfy the pre-condition of S and ρ_i, ρ_{i+1} and D satisfy its post-condition. For uniformity, blocking prefixes of runs are extended to infinite runs by repeating forever their last record.

The business process in the example exhibits a flexibility that, while desirable in practice for a postive customer experience, yields intricate runs, all of which need to be considered in verification. For instance, at any time before submitting a valid payment, the customer may edit the order (select a different product, shipping method, or change/add a coupon) an unbounded number of times. Likewise, the customer may cancel an order for a refund even after submitting a valid payment.

1.3 Temporal properties of artifact systems

The properties we are interested in verifying are expressed in a first-order extension of linear temporal logic called LTL-FO. This is a powerful language, fit to capture a wide variety of business policies implmented by a business process. For instance, in our running example it allows us to express such desiderata as:

> If a correct payment is submitted then at some time in the future either the product is shipped or the customer is refunded the correct amount.
>
> A free shipment coupon is accepted only if the available quantity of the product is greater than zero, the weight of the product is in the limit allowed by the shipment method, and the sum of price and shipping cost exceeds the coupon's minimum purchase value.

In order to specify temporal properties we use an extension of LTL (linear-time temporal logic). Recall that LTL is propositional logic augmented with temporal operators such as **G** (always), **F** (eventually), **X** (next) and **U** (until) (e.g., see [Pnu77]). For

example, $\mathbf{G}p$ says that p holds at all times in the run, $\mathbf{F}p$ says that p will eventually hold, and $\mathbf{G}(p \rightarrow \mathbf{F}q)$ says that whenever p holds, q must hold sometime in the future. The extension of LTL that we use, called[1] LTL-FO, is obtained from LTL by replacing propositions with quantifier-free FO statements about particular artifact records in the run. The statements use the artifact variables and may use additional *global* variables, shared by different statements and allowing to refer to values in different records. The global variables are universally quantified over the entire property.

For example, suppose we wish to specify the property that if a correct payment is submitted then at some time in the future either the product is shipped or the customer is refunded the correct amount. The property is of the form $\mathbf{G}(p \rightarrow \mathbf{F}q)$, where p says that a correct payment is submitted and q states that either the product is shipped or the customer is refunded the correct amount. Moreover, if the customer is refunded, the amount of the correct payment (given in p) should be the same as the amount of the refund (given in q). This requires using a global variable x in both p and q. More precisely, p is interpreted as the formula $\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}$ and q as $\texttt{status} = \text{"shipped"} \vee \texttt{amount\_refunded} = x$. This yields the LTL-FO property

$$(\varphi_1)\ \forall x \mathbf{G}((\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}) \rightarrow \mathbf{F}(\texttt{status} = \text{"shipped"} \vee \texttt{amount\_refunded} = x))$$

Note that, as one would expect, the global variable x is universally quantified at the end.

We say that an artifact system $\mathscr{A}$ satisfies an LTL-FO sentence φ if all runs of the artifact system satisfy φ for all values of the global variables. Note that the database is fixed for each run, but may be different for different runs.

We now show a second property φ_2 for the running example, expressed by the LTL-FO formula

$$(\varphi_2)\ \forall\, v,m,s,p,a,w,c,l(\mathbf{G}(\texttt{prod\_id} \neq \lambda \wedge \texttt{ship\_type} \neq \lambda \wedge \texttt{COUPONS}(\texttt{coupon}, \text{"free\_ship"}, v, m, s)) \wedge \texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge \texttt{SHIPPING}(\texttt{ship\_type}, c, l) \rightarrow \underbrace{a > 0}_{(i)} \wedge \underbrace{w \leq l}_{(ii)} \wedge \underbrace{p + c \geq m}_{(iii)})$$

[1] The variant of LTL-FO used here differs from previous ones in that the FO formulas interpreting propositions are quantifier-free. By slight abuse we use here the same name.

Property φ_2 verifies the consistency of orders that use coupons for free shipping. The premise of the implication lists the conditions for a completely specified order that uses such coupons. The conclusion checks the following business rules (i) available quantity of the product is greater than zero, (ii) the weight of the product is in the limit allowed by the shipment method, and (iii) the total order value satifies the minimum for the application of the coupon.

Note that this property holds only due to the integrity constraints on the schema. Indeed, observe that (i) is guaranteed by the post-condition of service **choose_product**, (ii) by **choose_shiptype**, and (iii) by **apply_coupon**. In the post-conditions, the checks are perfomed by looking up in the database the weight/price/cost/limit attributes associated to the customer's selection of product id and shipment type (stored in artifact variables). The property performs the same lookup in the database, and it is guaranteed to retrieve the same tuples only because product id and shipment type are keys for `PRODUCTS`, respectively `SHIPPING`. The verifier must take these key declarations into account, to avoid generating a spurious counter-example in which the tuples retrieved by the service post-conditions are distinct from those retrieved by the property, despite agreeing on product id and shipment type.

1.4 Automatic Verification of Artifact Systems

Let us consider first artifact systems and properties without arithmetic constraints or data dependencies. This case was studied in [DHPV09], with a slightly richer model in which artifacts can carry some limited relational state information (however, here we stick for simplicity to the earlier minimalistic model). The main result is the following.

Theorem 1.4.1. *It is decidable, given an artifact system $\mathscr{A}$ with no data dependencies or arithmetic constraints, and an LTL-FO property φ with no arithmetic constraints, whether $\mathscr{A}$ satisfies φ.*

The complexity of verification is PSPACE-complete for fixed-arity database and artifacts, and EXPSPACE otherwise. This is the best one can expect, given that even very simple static analysis problems for finite-state systems are already PSPACE-complete.

The main idea behind the verification algorithm is to explore the space of runs of the artifact system using *symbolic* runs rather than actual runs. This is based on the fact that the relevant information at each instant is the pattern of connections in the database between attribute values of the current and successor artifact records in the run, referred to as their *isomorphism type*. Indeed, the sequence of isomorphism types in a run can be generated symbolically and is enough to determine satisfaction of the property. Since each isomorphism type can be represented by a polynomial number of tuples (for fixed arity), this yields a PSPACE verification algorithm.

It turns out that the verification algorithm can be extended to specifications and properties that use a *total order* on the data domain, which is useful in many cases. This however complicates the algorithm considerably, since the order imposes global constraints that are not captured by the local isomorphism types. The algorithm was first extended in [DHPV09] for the case of a dense countable order with no end-points. This was later generalized to an arbitrary total order by Segoufin and Torunczyk [LS11] using automata-theoretic techniques. In both cases, the worst-case complexity remains PSPACE.

In Chapter 4, we provide an alternative way to perform verification on artifact systems with no dependencies and no arithmetic (albeit with no ordering), that is again in PSPACE but uses techniques similar to the ones used to verify systems with dependencies and arithmetic.

1.4.1 Verification with arithmetic constraints and data dependencies

Theorem 1.4.1 fails even in the presence of simple data dependencies or arithmetic. Specifically, as shown in [DHPV09, DDV11] and in Chapter **??**, verification becomes undecidable as soon as the database is equipped with at least one key dependency, *or* if the specification of the artifact system uses simple arithmetic constraints allowing to increment and decrement by one the value of some atributes. Therefore, a restriction is needed to achieve decidability. We discuss this next.

To gain some intuition, consider the undecidability of verification for artifact systems with increments and decrements. The proof of undecidability is based on the

ability of such systems to simulate *counter machines*, for which the problem of state reachability is known to be undecidable [Min67]. To simulate counter machines, an artifact system uses an attribute for each counter. A service performs an increment (or decrement) operations by "feeding back" the incremented (or decremented) value into the next occurrence of the corresponding attribute. To simulate counters, this must be done un unbounded number of times. To prevent such computations, the restriction imposed in Chapter 2 is designed to limit the data flow between occurrences of the same artifact attribute at different times in runs of the system that satisfy the desired property. As a first cut, a possible restriction would prevent any data flow path between unequal occurrences of the same artifact attribute. Let us call this restriction *acyclicity*. While acyclicity would achieve the goal of rendering verification decidable, it is too strong for many practical situations. In our running example, a customer can choose a shipping type and coupon and repeatedly change her mind and start over. Such repeated performance of a task is useful in many scenarios, but would be prohibited by acyclicity of the data flow. To this end, we define in Chapter 2 a more permissive restriction called *feedback freedom*. The formal definition considers, for each run, a graph capturing the data flow among variables, and imposes a restriction on the graph. Intuitively, paths among different occurrences of the same attribute are permitted, but only as long as each value of the attribute is independent on its previous values. This is ensured by a syntactic condition that takes into account both the artifact system and the property to be verified. We omit here the rather technical details. It is shown in Chapter 2 that feedback freedom of an artifact system together with an LTL-FO property can be checked in PSPACE by reduction to a test of emptiness of a two-way alternating finite-state automaton.

There is evidence ([VDATHKB03, AM00]) that the feedback freedom condition is permissive enough to capture a wide class of applications of practical interest. Indeed, this is confirmed by numerous examples of practical business processes modeled as artifact systems, encountered in our collaboration with IBM. Many of these, including typical e-commerce applications, satisfy the feedback freedom condition. The underlying reason seems to be that business processes are usually not meant to "chain" an unbounded number of tasks together, with the output of each task being input to the next. Instead, the unboundedness is usually confined to two forms, both consistent with

feeback-freedom:

1. Allowing a certain task to undo and retry an unbounded number of times, with each retrial independent of previous ones, and depending only on a context that remains unchanged throughout the retrial phase. A typical example is repeatedly providing credit card information until the payment goes through, while the order details remain unchanged. Another is the situation in which an order is filled according to sequentially ordered phases, where the customer can repeatedly change her mind within each phase while the input provided in the previous phases remains unchanged (e.g. changing her mind about the shipment type for the same product, the rental car reservation for the same flight, etc.)

2. Allowing a task to batch-process an unbounded collection of inputs, each processed independently, within an otherwise unchanged context (e.g. sending invitations to an event to all attendants on the list, for the same event details).

Feedback freedom turns out to ensure decidability of verification in the presence of arithmetic constraints, and also under a large class of data dependencies including key and foreign key constraints on the database. Thus, we prove in Chapter **??**:

Theorem 1.4.2. *It is decidable, given an artifact system $\mathscr{A}$ whose database satisfies a set of key and foreign key constraints, and an LTL-FO property φ such that $(\mathscr{A}, \varphi)$ is feedback free, whether every run of $\mathscr{A}$ on a valid database satisfies φ.*

The intuition behind decidability is the following. Recall the verification algorithm of Theorem 1.4.1. Because of the data dependencies and arithmetic constraints, the isomorphism types of symbolic runs are no longer sufficient, because every artifact record in a run is constrained by the entire history leading up to it. This can be specified as an $\exists$FO formula using one quantified variable for each artifact attribute occurring in the history, referred to as the *inherited constraint* of the record. The key observation is that due to feedback freedom, the inherited constraint can be rewritten into an $\exists$FO formula with quantifier rank[2] bounded by k^2, where k is the number of attributes of the artifact. This implies that there are only finitely many non-equivalent inherited constraints. This allows to use again a symbolic run approach to verification, by replacing

[2]The quantifier rank of a formula is the maximum number of quantifiers occurring along a path from root to leaf in the syntax tree of the formula, see [Lib04].

isomorphism types with inherited constraints.

1.4.2 Complexity

Unfortunately, there is a price to pay for the extension to data dependencies and arithmetic in terms of complexity. Recall that verification without arithmetic constraints or data dependencies can be done in PSPACE. In contrast, the worst-case complexity of the extended verification algorithm is very high – hyperexponential in k^2 (i.e. a tower of exponentials of height k^2). The complexity is more palatable when the clusters of artifact attributes that are mutually dependent are bounded by a small w. In this case, the complexity is hyperexponential in w^2. If the more stringent acyclicity restriction holds, the complexity drops to 2-EXPTIME in k.

In Chapter 4, we develop a series of restricted technique based on the concept of inherited constraint, in order to improve the worst-case complexity of the verification problem. A first improvement comes from limiting the data dependencies to just unary keys and considering no arithmetic. This results in an EXPSPACE upper bound w.r.t. the number of attributes, given a fixed-arity database. A nice result is also that this techniques allows us to relax the feedback freedom definition, in order to accept more specifications. Also, we identify a class of very common business process specification, defined by how joins are performed. Intuitively, if the joins happen only between attributes in a foreign key constraint, and foreign keys are acyclic in the database schema, the verification worst-case complexity becomes PSPACE, given a fixed-arity database. Even adding arithmetic, we are then able to extend the previous technique obtaining a worst-case complexity that is hyperexponential only in the maximum size of clusters of attributes that are mutually dependent with arithmetic operations. Again, we relax the feedback-freedom definition by considering separately database predicates and arithmetic operations.

1.5 Acyclic Workflows with Exceptions

In Chapter 5 we introduce a high-level business process model that relates closely to feedback-freedom. Also, the structure of these processes is based on pat-

terns found in the literature [VDATHKB03, RvdAtH, RHE05, RtHEvdA05] and are aimed to model realistic business processes. We call this model *Acyclic Workflows with Exceptions* (AWE) and it is fundamental in the developments of our feasibility tests. We specify the semantics of AWEs using our lower-level artifact model developed in Chapter 2.

Then, we compare the expressiveness of AWEs, our lower-level artifact model, and feedback-free systems. The important result of this section is that a natural restriction on AWEs implies feedback-freedom. This further strengthens the link between how business processes are specified (patterns) and feedback-freedom, and provides us a higher-level model to use in our feasibility study.

1.6 Feasibility study

We conducted a feasibility study of the verification techniques presented in Chapter 4. We implemented software prototype, described in detail in Chapter 6. Then, we perfomed a series experiments in order to establish the applicability of the techniques to realistic business processes. First, we created a random generator of business processes and temporal properties. This is based on patterns and frequencies extracted from real-world business process specifications [TLR07] and temporal properties [DAC98].

The complexity of the generated specifications is analyzed using the cyclomatic complexity metric [McC76], developed for traditional sequential programs but for which quality guidelines are available, and control flow complexity [Car05], developed specifically for business processes. These complexity measures are fundamental in gauging the effectiveness of our technique. Indeed, there are quality guidelines [McC76] that impose a certain maximum size of a business process. Any specification larger than a certain threshold is supposed to be broken up to retain efficiency, understandability and maitainability. Our technique has to be applicable to business processes more complex than the current thresholds.

In Chapter 7, we analyzed the running times of our verifier on the generated specifications, and we had running times varying from seconds to minutes for specifications significantly larger than the current threshold. For instance, a cyclomatic com-

plexity index of 50 is considered ‘untestable’, and our prototype verifies properties with complexity up to 80 in minutes.

Acknowledgement

Richard Hull, Alin Deutsch and Victor Vianu co-authored part of this chapter.

Part I

Theoretical Results

2 Model

2.1 Framework

The arithmetic constraints considered here are over domain $\mathbb{Q}$, the rational numbers. While databases could use non-numeric data, we assume for uniformity, and without loss of generality, that all structures are over $\mathbb{Q}$. We denote by $\mathscr{C}$ an infinite set of relation symbols, each of which has a fixed interpretation as the set of solutions of a finite set of linear inequalities with integer coefficients. By slight abuse, we sometimes use the same notation for a relation symbol in $\mathscr{C}$ and its fixed interpretation.

Definition 2.1.1. *An* artifact schema *is a a tuple $\mathscr{A} = \langle \bar{x}, \mathscr{D}\mathscr{B} \rangle$ where $\bar{x}$ is a finite set of* artifact variables *and $\mathscr{D}\mathscr{B}$ is a relational schema.*

For each $\bar{x}$, we also define a set of variables $\bar{x}' = \{x' \mid x \in \bar{x}\}$ where each x' is a distinct new variable.

Definition 2.1.2. *An* instance *of an artifact schema $\mathscr{A} = \langle \bar{x}, \mathscr{D}\mathscr{B} \rangle$ is a tuple $A = \langle \nu, D \rangle$ where ν is a valuation of $\bar{x}$ into $\mathbb{Q}$ and D is a finite instance of $\mathscr{D}\mathscr{B}$ whose domain is included in $\mathbb{Q}$.*

We denote by $\exists$FO the first-order formulas whose prenex form uses only existential quantification, and by $\mathrm{CQ}^{\neg}$ the formulas built from literals (positive and negated atoms over $\mathscr{D}\mathscr{B} \cup \mathscr{C} \cup \{=\}$) using only conjunction and existential quantification.

Definition 2.1.3. *A* service *over an artifact schema $\mathscr{A}$ is a pair $\sigma = \langle \pi, \psi \rangle$ where:*

- *$\pi(\bar{x})$, called* pre-condition*, is an $\exists$FO formula using relational symbols in $\mathscr{D}\mathscr{B} \cup \mathscr{C}$, with free variables $\bar{x}$;*

- $\psi(\bar{x},\bar{x}')$, *called* post-condition*, is an* $\exists$*FO formula on the relational symbols in* $\mathscr{D}\mathscr{B}\cup\mathscr{C}$*, with free variables* $\bar{x}\cup\bar{x}'$.

Definition 2.1.4. *An* artifact system *is a triple* $\Gamma = \langle\mathscr{A},\Sigma,\Pi\rangle$*, where* $\mathscr{A}$ *is an artifact schema,* Σ *is a non-empty set of services over* $\mathscr{A}$*, and* Π *is a pre-condition (as above, a* $\exists$*FO formula over* $\mathscr{D}\mathscr{B}\cup\mathscr{C}$*, with free variables* $\bar{x}$*).*

Definition 2.1.5. *Let* $\sigma = \langle\pi,\psi\rangle$ *be a service over an artifact schema* $\mathscr{A} = \langle\bar{x},\mathscr{D}\mathscr{B}\rangle$*, and let D be an instance over* $\mathscr{D}\mathscr{B}$*. Let* ν,ν' *be valuations of* $\bar{x}$*. We say that* ν' *is a* possible successor *of* ν *w.r.t.* σ *and D (denoted* $A \xrightarrow{\sigma} A'$ *when D is understood) iff:*

- $D\cup\mathscr{C} \models \pi(\nu)$*, and*
- $D\cup\mathscr{C} \models \psi(\nu,\nu')$.

Definition 2.1.6. *Let* $\Gamma = \langle\mathscr{A},\Sigma,\Pi\rangle$ *be an artifact system, where* $\mathscr{A} = \langle\bar{x},\mathscr{D}\mathscr{B}\rangle$*. A* run *of* Γ *on database instance D over* $\mathscr{D}\mathscr{B}$ *is an infinite sequence* $\rho = \{\rho_i\}_{i\geq 0}$ *of valuations of* $\bar{x}$ *so that* $D\cup\mathscr{C} \models \Pi(\rho_0)$ *and for each* $i \geq 0$*,* $\rho_i \xrightarrow{\sigma} \rho_{i+1}$ *for some* $\sigma \in \Sigma$*.*

We denote by $Runs_D(\Gamma)$ the set of all runs of Γ on database instance D.

Remark 2.1.7. *In [DHPV09], artifacts are equipped with state relations in addition to the database and artifact variables. However, under the guarded restriction, the state relations are essentially limited to be finite-state. Note that finite state control can be simulated with artifact variables, by having one variable hold the current state. For instance, this role is played by variable* status *in the example below. We therefore omit explicit states in the present model.*

We next illustrate the expressive power of the artifact system framework by modeling the running e-commerce example. Due to limited space, we only list some of the services involved.

Example 2.1.8 The running example models an e-commerce business process in which the customer chooses a product and a shipment method and applies various kinds of coupons to the order. There are two kinds of coupons: discount coupons subtract their value from the total (e.g. a $50 coupon) and free-shipment coupons subtract the shipping

costs from the total. The order is filled in a sequential manner (first pick the product, then the shipment, then claim a coupon), as is customary on e-commerce web-sites. After the order is filled, the system awaits for the customer to submit a payment. If the payment matches the amount owed, the system proceeds to shipping the product.

We define an artifact with the following variables:

status, prod_id, ship_type, coupon, amount_owed,
amount_paid, amount_refunded.

The status variable tracks the status of the order and can take the following values:

"edit_product", "edit_ship", "edit_coupon", "processing",
"received_payment", "shipping", "shipped", "canceling",
"canceled".

Artifact variables ship_type and coupon record the customer's selection, received as an external input. amount_paid is also an external input (from the customer, possibly indirectly via a credit card service). Variable amount_owed is set by the system using arithmetic operations that sum up product price and shipment cost, subtracting the coupon value. Variable amount_refunded is set by the system in case a refund is activated.

The database includes the following tables (underlined attributes denote keys):

PRODUCTS(<u>id</u>, price, availability, weight),
COUPONS(<u>code</u>, type, value, min_value, free_shiptype),
SHIPPING(<u>type</u>, cost, max_weight),
OFFERS(<u>prod_id</u>, discounted_price, active).

The database also satisfies the following foreign keys:

COUPONS[free_shiptype] $\subseteq$ SHIPPING[type] and
OFFERS[prod_id] $\subseteq$ PRODUCTS[id].

Our framework's domain is $\mathbb{Q}$, however, in order to enhance readability and without loss of generality, we allow non-numeric attributes over arbitrary domains, including in particular enumeration types (as for the `status` artifact variable).

The starting configuration has `status` initialized to "edit_prod", and all other variables to "undefined". By convention, in this example we model undefined variables using the reserved constant λ. (This is syntactic sugar and does not affect the artifact systems model. In the example for instance, any non-positive value can play this role.) The initialization is easily expressed by the artifact system's pre-condition Π.

The services. The following services model a few of the business process tasks.

choose_product The customer chooses a product.

π : `status` = "edit_prod"
ψ : $\exists p,a,w(\texttt{PRODUCTS}(\texttt{prod\_id}',p,a,w) \wedge a > 0)$
$\wedge \texttt{status}' =$ "edit_shiptype"

choose_shiptype The customer chooses a shipping option.

π : `status` = "edit_ship"
ψ : $\exists c,l,p,a,w(\texttt{SHIPPING}(\texttt{ship\_type}',c,l) \wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge l > w) \wedge$
$\texttt{status}' =$ "edit_coupon" $\wedge \texttt{prod\_id}' = \texttt{prod\_id}$

apply_coupon The customer optionally inputs a coupon number.

π : `status` = "edit_coupon"
ψ : $(\texttt{coupon}' = \lambda \wedge \exists p,a,w,c,l(\texttt{PRODUCTS}(\texttt{prod\_id},p,$
$a,w) \wedge \texttt{SHIPPING}(\texttt{ship\_type},c,l) \wedge \texttt{amount\_owed}'$
$= p+c) \wedge \texttt{status}' =$ "processing" $\wedge \texttt{prod\_id}' =$
$\texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type}) \vee$
$(\exists t,v,m,s,p,a,w,c,l(\texttt{COUPONS}(\texttt{coupon}',t,v,m,s) \wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge \texttt{SHIPPING}(\texttt{ship\_type},$
$c,l) \wedge p+c \geq m \wedge (t =$ "free_shipping" $\rightarrow$

$(s = \texttt{ship\_type} \wedge \texttt{amount\_owed}' = p)) \wedge$
$(t = \text{“discount”} \rightarrow \texttt{amount\_owed}' = p + c - v))$
$\wedge \texttt{status}' = \text{“processing”} \wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge$
$\texttt{ship\_type}' = \texttt{ship\_type})$

Notice that the pre-conditions π of the services check the value of the `status` variable. For instance, according to **choose_product**, the customer can only input her product choice while the order is in “edit_prod” status.

Also notice that the post-conditions ψ constrain the next values of the artifact variables (denoted by a prime). For instance, according to **choose_product**, once a product has been picked, the next value of the status variable is “edit_shiptype”, which will at a subsequent step enable the **choose_shiptype** service (by satisfying its pre-condition). Similarly, once the shipment type is chosen (as modeled by service **choose_shiptype**), the new status is “edit_coupon”, which enables the **apply_coupon** service. The interplay of pre- and post-conditions achieves a sequential filling of the order, starting from the choice of product and ending with the claim of a coupon.

A post-condition may refer to both the current and next values of the artifact variables. For instance, in service **choose_shiptype**, the fact that only the shipment type is picked while the product remains unchanged, is modeled by preserving the product id: the next and current values of the corresponding artifact variable are set equal.

Pre- and post-conditions may query the database. For instance, in service **choose_product**, the post-condition ensures that the product id chosen by the customer is that of an available product (by checking that it appears in a `PRODUCTS` tuple, whose availability attribute is positive).

Finally, notice the arithmetic computation in the post-conditions. For instance, in service **apply_coupon**, the sum of the product price p and shipment cost c (looked up in the database) is adjusted with the coupon value (notice the distinct treatment of the two coupon types) and stored in the `amount_owed` artifact variable.

Observe that the first post-condition disjunct models the case when the customer inputs no coupon number (the next value $\texttt{coupon}'$ is set to undefined), in which case a different owed amount is computed, namely the sum of price and shipping cost. □

Dependencies We consider integrity constraints of the form

$$\forall \bar{u}\bar{w}\ \phi(\bar{u},\bar{w}) \rightarrow \exists \bar{v}\, \psi(\bar{u},\bar{v})$$

where ϕ and ψ are conjunctions of relational and equality atoms (positive literals over the vocabulary including the relational schema and the equality predicate, respectively). Such sentences are known as *embedded dependencies* and are sufficiently expressive to specify all usual integrity constraints, such as keys, foreign keys, inclusion, join, multivalued dependencies, etc. [Fag82, AHV95]. In this paper, we refer to embedded dependencies in short as "dependencies". We call ϕ the *premise* and ψ the *conclusion*. We write $A \models \Sigma$ if the instance A satisfies all the dependencies in Σ.

Example 2.1.9 We illustrate dependencies continuing Example 2.1.8. The key constraint`PRODUCTS` is expressed by the dependency

$$\begin{aligned}&\forall i, p_1, a_1, w_1, p_2, a_2, w_2\\ &\qquad \texttt{PRODUCTS}(i,p_1,a_1,w_1) \wedge \texttt{PRODUCTS}(i,p_2,a_2,w_2)\\ &\qquad \rightarrow p_1 = p_2 \wedge a_1 = a_2 \wedge w_1 = w_2.\end{aligned}$$

The foreign key constraint on the `OFFERS` table is expressed by

$$\forall i,d,v\ \texttt{OFFERS}(i,d,v) \rightarrow \exists p,a,w\ \texttt{PRODUCTS}(i,p,a,w). \qquad \square$$

2.2 Temporal properties of artifact systems

In order to specify temporal properties we use an extension of LTL (linear-time temporal logic). Recall that LTL is propositional logic augmented with temporal operators X (next) and U (until) (e.g., see [Pnu77]). The extension we use, called[1] LTL-FO, is obtained from LTL by interpreting propositions as FO statements about particular artifact instances in the run. The different statements may share some global variables, that are universally quantified.

[1]The variant of LTL-FO used here differs from previous ones in that the FO formulas interpreting propositions are quantifier-free. By slight abuse we use here the same name.

Definition 2.2.1. *Let $\mathscr{A} = \langle \bar{x}, \mathscr{D}\mathscr{B} \rangle$ be an artifact schema. An* FO component *over $\mathscr{A}$ is a quantifier-free FO formula over $\mathscr{D}\mathscr{B} \cup \mathscr{C}$. An LTL-FO formula over $\mathscr{A}$ is an expression $\forall \bar{y} \varphi_f$, where:*

(i) φ is an LTL formula with propositions P;

(ii) f is a mapping from P to FO components over $\mathscr{A}$

(iii) φ_f is obtained by replacing each $p \in P$ with $f(p)$;

(iv) $\bar{y}$ is the set of variables occurring in φ_f that are different from $\bar{x} \cup \bar{x}'$.

The semantics of LTL-FO formulas is defined as follows. Let $\langle \mathscr{A}, \Sigma, \Pi \rangle$ be an artifact system, $\forall \bar{y} \varphi_f$ an LTL-FO formula over $\mathscr{A}$, and ρ a run of $\langle \mathscr{A}, \Sigma, \Pi \rangle$ on database D. Let μ be a valuation of $\bar{y}$ into $\mathbb{Q}$. An FO component $\psi(\bar{x}, \bar{x}', \bar{y})$ of φ_f is satisfied in ρ_i with valuation μ if $D \cup \mathscr{C} \models \psi(\rho_i, \rho_{i+1}, \mu)$, $i \geq 0$. The run ρ satisfies φ_f with valuation μ if $\{\sigma(\rho_i)\}_{i \geq 0} \models \varphi$, where $\sigma(\rho_i)$ is the truth assignment for P in which p is true iff $f(p)$ is satisfied in ρ_i with valuation μ. Finally, $\rho \models \forall \bar{y} \varphi_f$ if $\rho \models \varphi_f$ with every valuation μ of $\bar{y}$ into $\mathbb{Q}$.

We say that an artifact system Γ satisfies an LTL-FO sentence φ, denoted $\Gamma \models \varphi$, if all runs of Γ satisfy φ. Note that the database is fixed for each run, but may be different for different runs.

We illustrate LTL-FO in the context of Example 2.1.8.

Example 2.2.2 We show a few properties that specify desirable business rules for the running example.

$$(\varphi_1)\ \forall x \mathbf{G}((\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}) \\ \rightarrow \mathbf{F}(\texttt{status} = \text{"shipped"} \vee \texttt{amount\_refunded} = x))$$

Property φ_1 states that if a correct payment is submitted then at some time in the future either the product is shipped or the customer is refunded the correct amount. φ_1 is obtained from LTL property $\varphi = \mathbf{G}(p \rightarrow \mathbf{F}q)$ via the mapping f_1, where $f_1(p) =$ $\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}$ and $f_1(q) = \texttt{status} = \text{"shipped"} \vee$ $\texttt{amount\_refunded} = x$. Note the use of universally-quantified variable x to relate the value of paid and refunded amounts across distinct steps in the run sequence.

$$(\varphi_2)\ \forall\, v,m,s,p,a,w,c,l(\mathbf{G}(\texttt{prod\_id} \neq \lambda \wedge \texttt{ship\_type} \neq \lambda$$
$$\wedge \texttt{COUPONS}(\texttt{coupon}, \text{"free\_ship"}, v, m, s)) \wedge$$
$$\texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge \texttt{SHIPPING}(\texttt{ship\_type}, c, l)$$
$$\rightarrow \underbrace{a > 0}_{(i)} \wedge \underbrace{w \leq l}_{(ii)} \wedge \underbrace{p + c \geq m}_{(iii)})$$

Property φ_2 verifies the consistency of orders that use coupons for free shipping. The premise of the implication lists the conditions for a completely specified order that uses such coupons. The conclusion checks the following business rules (i) available quantity of the product is greater than zero, (ii) the weight of the product is in the limit allowed by the shipment method, and (iii) the total order value satifies the minimum for the application of the coupon.

Note that this property holds only due to the integrity constraints on the schema. Indeed, observe that (i) is guaranteed by the post-condition of service **choose_product**, (ii) by **choose_shiptype**, and (iii) by **apply_coupon**. In the post-conditions, the checks are perfomed by looking up in the database the weight/price/cost/limit attributes associated to the customer's selection of product id and shipment type (stored in artifact variables). The property performs the same lookup in the database, and it is guaranteed to retrieve the same tuples only because product id and shipment type are keys for `PRODUCTS`, respectively `SHIPPING`. The verifier must take these key declarations into account, to avoid generating a spurious counter-example in which the tuples retrieved by the service post-conditions are distinct from those retrieved by the property, despite agreeing on product id and shipment type. □

We note right away that one can easily eliminate the global variables $\bar{y}$ of the LTL-FO formula $\forall \bar{y} \varphi_f$.

Lemma 2.2.3. *Given* Γ *and* $\forall \bar{y} \varphi_f$ *as above, one can construct in linear time an artifact system* Γ' *such that* $\Gamma \models \forall \bar{y} \varphi_f$ *iff* $\Gamma' \models \varphi_f$.

Indeed, Γ' is obtained from Γ by simply adding $\bar{y}$ to its artifact variables and propagating their values at each transition. Thus,we can assume that the LTL-FO formulas to be verified have no global variables. Clearly, $\Gamma \models \varphi_f$ iff there is no run of Γ satisfying $\neg \varphi_f$. The verification problem will focus on the latter formulation.

Not surprisingly, model checking is undecidable for artifact systems and LTL-FO properties, even if the system uses only a database (satifying given FDs) and no arithmetic constraints, or only arithmetic constraints and no database.

Theorem 2.2.4. *(i) It is undecidable, given an artifact system* $\Gamma = \langle \mathscr{A}, \Sigma, \Pi \rangle$ *where* $\mathscr{A} = \langle \bar{x}, \mathscr{DB} \rangle$*, a set* F *of FDs over* $\mathscr{DB}$*, and an LTL-FO property* φ *such that* φ *and all pre-and-post conditions of* Σ *and* Π *use only relations in* $\mathscr{DB}$*, whether* φ *holds for all runs of* Γ *on databases satisfying* F*.*

(ii) It is undecidable, given an artifact system $\Gamma = \langle \mathscr{A}, \Sigma, \Pi \rangle$ *where* $\mathscr{A} = \langle \bar{x}, \mathscr{DB} \rangle$ *and an LTL-FO property* φ *such that* φ *and all pre-and-post conditions of* Σ *and* Π *use only constraints in* $\mathscr{C}$*, whether* $\Gamma \models \varphi$*.*

Proof: Part (i) follows from Theorem 4.2 in [DHPV09]. Part (ii) is shown by defining an artifact system that simulates a counter machine in conjunction with a property that further forbids reachability of a given state of the machine (details ommitted). Since state reachability is undecidable for counter machines [Min67], the result follows. □

Remark 2.2.5. *Note that, in the absence of dependencies and arithmetic, the artifact systems discussed here fall in the class of "guarded" artifact systems introduced in [DHPV09] towards decidable static verification. Theorem 2.2.4 shows that the results of [DHPV09] do not transfer to the new setting. As detailed below, overcoming the technical challenges introduced by arithmetic and dependencies requires developing a fundamentally different proof technique and a novel syntactic restriction that yields decidability.*

2.3 Feedback-free artifact systems

We next define the feedback-free syntactic restriction, applying jointly to an artifact system and a property to be verified. The original intuition behind the notion of feedback freedom comes from the proof of Theorem 2.2.4(ii), which shows that artifact systems can simulate counter machines. Such an artifact system uses a service that performs the increment operation on a counter variable, and allows the run to "feed back" the incremented value into the same service (by updating the same counter variable) for

an unbounded number of times. Decrement is handled similarly. This simulation of unbounded counters is responsible for undecidability. The feedback-freedom restriction is designed to limit the data flow between occurrences of the same artifact variable at different times in runs of the system that satisfy the desired property. This precludes the ability to perform the kind of unbounded computations needed to simulate counter machines. The intuition is further discussed after the formal definition of feedback-freedom.

Symbolic runs In order to formalize the feedback free condition, we use the notion of *symbolic run*. This will also provide the central component of our verification algorithm presented in the next section.

Let $\Gamma = \langle \mathscr{A}, \Sigma, \Pi \rangle$ be an artifact system, where $\mathscr{A} = \langle \bar{x}, \mathscr{D}\mathscr{B} \rangle$, and φ_f be an LTL-FO formula with no global variables. Recall that our aim is to develop a procedure for checking whether there exists a run of Γ satisfying $\neg\varphi_f$.

To each $x \in \bar{x}$ we associate an infinite set of new variables $\{x_i\}_{i>0}$, and we denote $\bar{x}_i = \{x_i \mid x \in \bar{x}\}$. A symbolic run ρ consists of a sequence $\{\psi_i(\bar{x}_i, \bar{x}_{i+1})\}_{i\geq 0}$ where each $\psi_i(\bar{x}_i, \bar{x}_{i+1})$ is a CQ$^\neg$ formula over $\mathscr{A}$ with free variables among $\bar{x}_i \cup \bar{x}_{i+1}$. The formulas ψ_i are obtained from Σ and φ_f as follows. We first define the sets of CQ$^\neg$ formulas below, that capture symbolically the possible transitions in Γ, together with truth assignments to the propositions in φ, expanded in φ_f via f. As earlier, P denotes the set of propositions in φ.

1. Δ_Σ. For each service $\langle \pi, \psi \rangle \in \Sigma$, consider the formula $\exists \bar{z} \xi$ obtained from $\pi \wedge \psi$ by putting it in prenex form and its quantifier-free body in DNF. For each such formula, Δ_Σ contains all formulas of the form $\exists \bar{z} \xi_d$, where ξ_d is a disjunct of ξ.

2. Δ_{φ_f} containing, for each $\sigma \in 2^P$, all disjuncts of the DNF of the formula

$$\bigwedge_{\sigma(p)=1} f(p) \wedge \bigwedge_{\sigma(p)=0} \neg f(p).$$

A *symbolic transition template* is a conjunction $\psi(\bar{x}, \bar{x}')$ of one formula from Δ_Σ and one from Δ_{φ_f}. Intuitively, the formula chosen from Δ_Σ corresponds to a transition caused by one of the services in Σ, while the formula chosen from Δ_{φ_f} determines a

truth assignment σ for the FO components of φ_f. Note that there are finitely many such formulas associated with Δ_Σ and Δ_{φ_f}. For $i > 0$, each formula ψ_i in the symbolic run is obtained from some symbolic transition template $\psi(\bar{x},\bar{x}')$ by replacing $\bar{x}$ with $\bar{x}_i$ and $\bar{x}'$ with $\bar{x}_{i+1}$. We refer to ψ_i as a *symbolic transition* generated by ψ. For $i = 0$, ψ_0 is obtained by taking the conjunction of a formula obtained as above with a formula accounting for the pre-condition Π (specifically, a disjunct of the DNF of Π in prenex form, where $\bar{x}$ is replaced with $\bar{x}_0$). We denote by σ_i the truth assignment to the propositions P of φ defined by $\sigma_i(p) = \sigma(f(p))$. We say that the symbolic run $\rho = \{\psi_i\}_{i\geq 0}$ satisfies $\neg\varphi_f$, denoted $\rho \models \neg\varphi_f$, iff $\{\sigma_i\}_{i\geq 0}$ satisfies $\neg\varphi$.

To formalize the feedback-free condition, we associate two undirected graphs G_ψ and E_ψ to each symbolic transition template $\psi = \exists\bar{z}(\phi(\bar{x},\bar{x}',\bar{z}))$. The graph G_ψ captures all connections among variables occurring together in the same atom, whereas E_ψ captures equalities alone. Specifically, G_ψ consists of the restriction to $\bar{x},\bar{x}'$ of the transitive closure of the graph containing an edge among every two variables occurring together in an atom of ψ, and E_ψ is the restriction to $\bar{x},\bar{x}'$ of the transitive closure of the graph containing an edge among every two variables in ψ that appear together in an equality atom of ψ.

Similarly, we define for each symbolic transition ψ_i the graphs E_{ψ_i} and G_{ψ_i} by replacing $\bar{x}$ by $\bar{x}_i$ and $\bar{x}'$ with $\bar{x}_{i+1}$ in E_ψ and G_ψ. Given a symbolic run $\rho = \{\psi_i\}_{i\geq 0}$, we define $G_\rho = \cup_{i\geq 0} G_{\psi_i}$ and E_ρ as $\cup_{i\geq 0} E_{\psi_i}$. We also denote by E^*_ρ the transitive closure of E_ρ. The graphs associated with finite symbolic runs are defined analogously.

Clearly, E^*_ρ is an equivalence relation on the variables of ρ. For each variable x_i, we denote by $[x_i]$ its equivalence class with respect to E^*_ρ. The *span* of an equivalence class $[x_i]$ is defined as $span([x_i]) = \{j \mid x_j \in [x_i]\}$. It is clear that $span([x_i])$ is always an interval (possibly infinite).

Example 2.3.1 We illustrate symbolic transition templates and the associated graphs $G_\psi, E_\psi, G_\rho, E_\rho$ using the artifact system from Example 2.1.8, and the following property

$$\varphi_f = \mathbf{F}(\texttt{status} = \text{“shipped”} \vee \texttt{status} = \text{“canceled”}).$$

The corresponding Δ_{φ_f} is

$\Delta_{\varphi_f} = \{\texttt{status} = \text{“shipped”} \vee \texttt{status} = \text{“canceled”},$

$\neg(\texttt{status} = \text{“shipped”} \vee \texttt{status} = \text{“canceled”})\}$.

To build Δ_Σ, we need to rewrite the conjunction of pre- and post-condition for each service into prenex DNF, and add each disjunct as a separate formula to Δ_Σ. The pre- and post-conditions of services **choose_product** and **choose_shiptype** are conjunctive, so only trivial prenex normal form rewriting is needed, which we omit. For service **apply_coupon**, we obtain five disjuncts $\bigvee_{i=1}^{5} \xi_i$ for the DNF of $\pi \wedge \psi$. We show ξ_2 below; ξ_1 corresponds to the case when the customer inputs no coupon number, while $\xi_2, \ldots, \xi_5$ correspond to various coupon type combinations (discount, free shipping). ξ_2 is the case of a discount coupon of value v:

$$
\begin{aligned}
\xi_2\text{: } & \texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \\
& \wedge\ \texttt{status} = \text{“edit\_coupon”} \wedge \texttt{status}' = \text{“processing”} \wedge \\
& \exists\, t, v, m, s, p, a, w, c, l\ (\\
& \quad \texttt{COUPONS}(\texttt{coupon}', t, v, m, s) \wedge \texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \\
& \quad \wedge \texttt{SHIPPING}(\texttt{ship\_type}, c, l) \wedge \neg(t = \text{“free\_shipping”}) \\
& \quad \wedge\ p + c \geq m \wedge\ \texttt{amount\_owed}' = p + c - v)
\end{aligned}
$$

Given the above sets $\Delta_\Sigma, \Delta_{\varphi_f}$, one of the resulting symbolic transition templates is for instance

$$\psi = \xi_2 \wedge \neg(\texttt{status} = \text{“shipped”} \vee \texttt{status} = \text{“canceled”}).$$

Figure 6.1(a) depicts the graphs G_ψ and E_ψ for all transition templates. In the case of **apply_coupon**, we indicate which disjunct ξ_i is used. The dashed (blue) edges correspond to equality atoms and belong to both E_ψ and G_ψ. Solid (red) edges correspond to relational atoms and belong to G_ψ only.

For instance, in the case of **choose_shiptype**, the dashed edge from `prod_id` to $\texttt{prod\_id}'$ reflects the fact that the product id remains unchanged, as specified by the corresponding equality atom in the post-condition. The solid edge from `prod_id` to `ship_type` reflects the folowing transitive connection between the two artifact variables: `prod_id` is directly connected to non-artifact variable w by co-occurrence in the `PRODUCTS` atom; w is directly connected to l via the arithmetic constraint; l is directly connected to $\texttt{ship\_type}'$ via the `SHIPPING` atom.

Since the `status` artifact variable does not appear in any constraint with another variable (neither in the services nor in the property), it does not affect the graphs of the symbolic transition templates, and it is dropped to avoid clutter. For the same reason we do not show the entire transitive closure of edges for either the G_ψ and E_ψ graphs.

Other services include **edit_product, edit_shiptype** and **edit_coupon**, whose specification we omit. However, the graphs already show that they perform expected tasks: when the user edits a coupon, product and shipment type are preserved (as depicted by the two dashed edges); when the shipment type is edited, only the product id is preserved, coupon information is forgotten, requiring subsequent re-input once the shipment type is known; finally, when the product id is edited, nothing is preserved, as new shipment type and coupon information will need to be re-input subsequently.

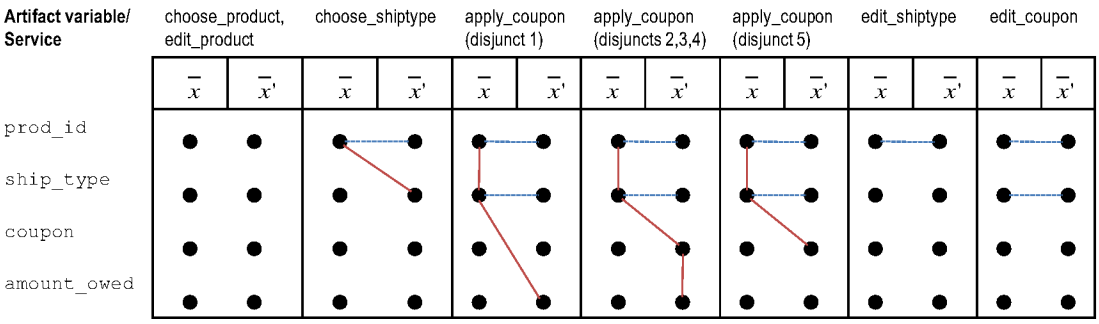

Figure 2.1: Graphs G_ψ and E_ψ for the symbolic transition templates in Example 2.3.1

Now consider a run ρ whose prefix corresponds to the following sequence of events: the customer chooses a product, then a shipment type, then applies a discount coupon, then changes her mind and edits the shipment type, chooses a new shipment type, and applies a discount coupon again. A resulting computation graph G_ρ and its restriction to equality edges E_ρ are depicted in Figure 2.2. The parenthesis annotating service **apply_coupon** means that disjunct ξ_2 is used, since it corresponds to a free shipping coupon. Notice that G_ρ (E_ρ) is obtained as the concatenation of the corresponding G_ψ (E_ψ) graphs from Figure **??**.

□

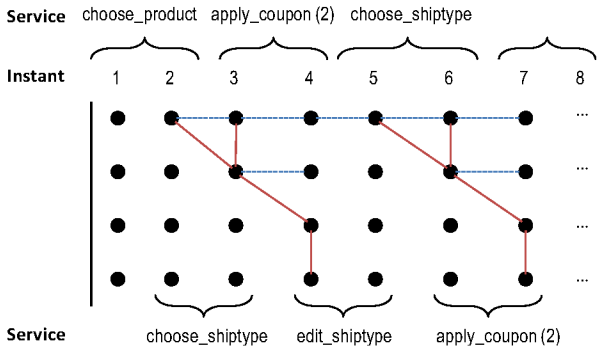

Figure 2.2: Computation graph G_ρ for Example 2.3.1

We now define feedback-freedom. We assume the notation developed above.

Definition 2.3.2. (Γ, φ_f) *is* feedback-free *iff for every symbolic run prefix* $\rho = \{\psi_i\}_{i \leq n}$, *each path from* x_i *to* x_j *in* G_ρ *contains a node* y *such that* $span([x_i]) \cup span([x_j]) \subseteq span([y])$.

By extension, we say that $(\Gamma, \forall \bar{y} \varphi_f)$ is feedback-free if (Γ', φ_f) is feedback-free, with Γ' obtained from $(\Gamma, \forall \bar{y} \varphi_f)$ as in Lemma 2.2.3.

Example 2.3.3 We illustrate some of the checks required to establish feedback freedom for the artifact system in the running example, on the symbolic run ρ of Example 2.3.1 (we cannot, of course, enumerate all runs).

Let `prod_id` play the role of y from Definition 2.3.2, `ship_type` that of x, and instants 3 and 7 the roles of i, respectively j.

Consider the graph G_ρ shown in Figure 6.1(b), and nodes $\texttt{ship\_type}_3$ and $\texttt{ship\_type}_7$, corresponding to artifact variable `ship_type` at instants 3 and 7. Recall that the dashed (blue) edges depict equality atoms, and connected components thereof repesent equivalence classes. We have :

$[\texttt{ship\_type}_3] = \{\texttt{ship\_type}_3, \texttt{ship\_type}_4\}$,
$span([\texttt{ship\_type}_3]) = [3,4]$,
$[\texttt{ship\_type}_7] = \{\texttt{ship\_type}_7, \texttt{ship\_type}_6\}$,
$span([\texttt{ship\_type}_7]) = [6,7]$.

Notice that every path from $\texttt{ship\_type}_3$ to $\texttt{ship\_type}_7$ must pass through node $\texttt{prod\_id}_4$, and that

$$span([\texttt{prod\_id}_4]) \supseteq [2,7]$$
$$\supseteq span([\texttt{ship\_type}_3]) \cup span([\texttt{ship\_type}_7]).$$

□

As discussed earlier, feeback-freedom is meant to restrict the ability to perform computation such as needed to simulate a counter machine, requiring repeated increments/decrements of the same variable. This is done by preventing unbounded updates to a variable's current value that depend on its history. Instead, while feedback-free processes still support updating an artifact variable (x) an unbounded number of times, feedback-freedom guarantees that each updated value is independent of how the value of x evolved historically from step i to step j ($i < j$). Indeed, x_j depends only on the values of *other* variables (say y), which are preserved throughout the computation from i to j ($span([y]) \supseteq span([x_i]) \cup span([x_j])$).

Example 2.3.4 In Example 2.3.3, the arithmetic constraint satisfied by the shipment type considered at instant 7 does not depend on the previous shipment choice at instant 3, and can be described directly in relation to the product id, which remains constant throughout. This independence would hold even in a run in which the customer repeatedly alternated between making up and changing her mind about the shipment type, possibly re-considering (and again discarding) the same shipment types several times. Similarly, the current balance can be computed directly on the current order snapshot, being independent of the previous ones. □

We formalize this intuition in Section 3.1, where we show that under feedback-freedom it suffices for the verification algorithm to keep only a "compressed" history of bounded size, in form of "inherited constraints" on the artifact variable values at every step.

We claim that the feedback freedom condition is permissive enough to capture a wide class of applications of practical interest. Indeed, this is confirmed by numerous

examples of practical business processes modeled as artifact systems, that we encountered due to our collaboration with IBM. Many of these, including typical e-commerce applications, satisfy the feedback freedom condition. The underlying reason seems to be that business processes are usually not meant to "chain" an unbounded number of tasks together, with the output of each task being input to the next. Instead, the unboundedness is usually confined to two forms, both consistent with feeback-freedom:

1. Allowing a certain task to undo and retry an unbounded number of times, with each retrial independent of previous ones, and depending only on a context that remains unchanged throughout the retrial phase. A typical example is repeatedly providing credit card information until the payment goes through, while the order details remain unchanged. Another is the situation in which an order is filled according to sequentially ordered phases, where the customer can repeatedly change her mind within each phase while the input provided in the previous phases remains unchanged (e.g. changing her mind about the shipment type for the same product, the rental car reservation for the same flight, etc.)

2. Allowing a task to batch-process an unbounded collection of inputs, each processed independently, within an otherwise unchanged context (e.g. sending invitations to an event to all attendants on the list, for the same event details).

A way to visualize the types of unboundedness represented in business processes is the following. Consider a *call graph* whose nodes are the actual invocations of services during the run (with instantiated input parameters). Draw an edge from task invocation a to task invocation b whenever b takes as input some of a's output, or reads global data modified by a. The call graph is (or can be unfolded into) a tree. Feedback-free processes disallow call trees of unbounded height, but allow unbounded outdegrees of the tree nodes.

Testing feedback-freedom We next show that testing feedback-freedom of (Γ, φ_f) can be reduced to testing emptiness of a non-deterministic finite-state automaton $A_{(\Gamma,\varphi_f)}$. Since a direct construction requires some rather tedious bookkeeping, we describe instead for simplicity a *two-way* alternating automaton $T_{(\Gamma,\varphi_f)}$ whose emptiness is equivalent to feedback-freedom. Recall that, in each transition, the head of a two-way automaton may move to the right or to the left of the current position, or may be stationary. An

alternating automaton augments non-determinism with universal and existential states. Acceptance from an existential state occurs if there *exists* a transition to a next state leading to acceptance, whereas acceptance from a universal state occurs if *all* transitions to a next state lead to acceptance. It is well-known that two-way alterating automata can be converted to classical one-way automata, and testing emptiness is PSPACE-complete (see [MNG08, LLS84, Bir96]).

Intuitively, $T_{(\Gamma,\varphi_f)}$ guesses a violation of feedback-freedom for (Γ,φ_f). Recall that a violation consists in a prefix ρ of a symbolic run in which there are two occurrences x_i, x_j of the same variable x in configurations i and j such that there exists a path from x_i to x_j in G_ρ so that no variable v occurring along the path has the property that $span(x_i) \cup span(x_j) \subseteq span(v)$.

We next describe $T_{(\Gamma,\varphi_f)}$ informally. The alphabet consists of the symbolic transition templates, augmented with several kinds of markers:

- ε (emtpy marker)
- x^1 for $x \in \bar{x}$; this identifies one occurrence of x
- x^2 for $x \in \bar{x}$; this identifies a second occurrence of x
- $[^1_x$ and $]^1_x$ for $x \in \bar{x}$; this identifies the beginning, resp. the end of the span of x^1;
- $[^2_x$ and $]^2_x$ for $x \in \bar{x}$; this identifies the beginning, resp. the end of the span of x^2;

Formally, the alphabet of $T_{(\Gamma,\varphi_f)}$ consists of pairs of symbolic transition templates and subsets of the above markers. The automaton $T_{(\Gamma,\varphi_f)}$ performs the following tests, carried out sequentially in multiple passes. For an occurrence of a marker α, we denote by $pos(\alpha)$ its position in the word.

- there is a single occurrence of each of the markers $x^i, [^i_x,]^i_x$ for some $x \in \bar{x}$ and $pos([^i_x) \leq pos(x^i) \leq pos(]^i_x)$, $i \in \{1,2\}$;
- $span(x_i) = [pos([^i_x), pos(]^i_x)]$; this can be tested by first generating a path in E_ρ from x in $pos(x^i)$ to some variable in $pos([^i_x)$, and similarly a path from x in $pos(x^i)$ to some variable in $pos(]^i_x)$, and then checking that *every* non-deterministically generated path in E_ρ originating from x in $pos(x^i)$ never reaches $pos([^i_x) - 1$ or $pos(]^i_x) + 1$. The latter is easily done using universal states.

- there exists a path from x in $pos(x^1)$ to x in $pos(x^2)$ in G_ρ such that for every variable v along the path, $span(v)$ does not include $[min\{pos([^i_x) \mid i = 1,2\}, max\{pos(]^i_x) \mid i = 1,2\}]$. This is checked using an alternation of existential and universal states. Specifically, every time a new v is generated along the path, the following must be tested for *every* path in E_ρ starting at v: the path does not touch both $min\{pos([^i_x) \mid i = 1,2\}$ and $max\{pos(]^i_x) \mid i = 1,2\}]$.

It is easily seen that one can construct a two-way alternating automaton testing the above properties, that uses polynomially many states in (Γ, φ_f). However, the automaton uses an alphabet of exponential size with respect to (Γ, φ_f), because there are exponentially many symbolic transition templates. Observe that the size of *each* alphabet symbol is polynomial in (Γ, φ_f).

We have the following:

Lemma 2.3.5. *(Γ, φ_f) is feedback-free iff the language accepted by $T_{(\Gamma,\varphi_f)}$ is empty.*

As stated earlier, emptiness of two-way alternating automata can be checked in PSPACE (see [MNG08, LLS84, Bir96]) with respect to the number of states and the size of the alphabet. Recall that the size of the alphabet of $T_{(\Gamma,\varphi_f)}$ is exponential with respect to (Γ, φ_f). However, because the reduction in [Bir96] yields an exponential blowup of the input in the number of states but not in the size of the alphabet, it follows that the complexity of testing emptiness of $T_{(\Gamma,\varphi_f)}$ remains PSPACE with respect to (Γ, φ_f). This completes the proof.

Remark 2.3.6. *The automata-theoretic approach to testing feedback-freedom can be used to refine the notion of feedback-free by taking into account additional restrictions on the allowed runs of the artifact system. For example, if additional control is specified by a Büchi automaton $\mathscr{B}$, testing feedback-freedom for runs satisfying the additional control reduces to testing emptiness of the cross-product automaton $\mathscr{B} \times A_{(\Gamma,\varphi_f)}$ (with $A_{(\Gamma,\varphi_f)}$ easily turned first into a Büchi automaton).*

2.4 Full E-Commerce Example

We model a simplified e-commerce business process. The customer can choose a product, a shipment method and apply a coupon to the order. There are two kinds of coupons: discount coupons subtract their value from the total (e.g. a $50 coupon) and free-shipment coupons subtract the shipping costs from the total.

The order is filled in a sequential manner as is customary on e-commerce websites. After the order is filled, the system awaits for the customer to submit a payment. If the payment matches the amount owed, the system proceeds to shipping the product. At any time before submitting a valid payment, the customer may edit the order (select a different product, shipping method, or change/add a coupon) an unbounded number of times. At any time, even after submitting a valid payment, the customer may cancel the order, for a refund, if the order hasn't shipped yet.

The artifact variables. We model our running example using an artifact system. We define an artifact with the following variables:

`status`, `prod_id`, `ship_type`, `coupon`, `amount_owed`,
`amount_paid`, `amount_refunded`.

The `status` variable tracks the status of the order and can take the following values:

"edit_product", "edit_ship", "edit_coupon", "processing", "received_payment", "shipping", "shipped", "canceling", "canceled".

Artifact variables `ship_type` and `coupon` record the customer's selection, received as an external input. `amount_paid` is also an external input (from the customer, possibly indirectly via a credit card service). Variable `amount_owed` is set by the system using arithmetic operations that sum up product price and shipment cost, subtracting the coupon value. Variable `amount_refunded` is set by the system in case a refund is activated.

The database includes the following tables (underlined attributes denote keys):

PRODUCTS(id, price, availability, weight),

COUPONS(code, type, value, min_value, free_shiptype),

SHIPPING(type, cost, max_weight),

OFFERS(prod_id, discounted_price, active).

The database also satisfies the following foreign keys:

COUPONS[free_shiptype] $\subseteq$ SHIPPING[type] and

OFFERS[prod_id] $\subseteq$ PRODUCTS[id].

Our framework's domain is $\mathbb{Q}$, however, in order to enhance readability and without loss of generality, we allow non-numeric attributes over arbitrary domains, including in particular enumeration types (as for the status artifact variable).

The starting configuration has status initialized to "edit_prod", and all other variables to "undefined". By convention, in this example we model undefined variables using the reserved constant λ. (This is syntactic sugar and does not affect the artifact systems model. In the example for instance, any non-positive value can play this role.) This is easily expressed by the artifact system's pre-condition Π:

$$\begin{aligned}\Pi \quad : \quad & \texttt{status} = \text{"edit\_prod"} \wedge \texttt{prod\_id} = \lambda \wedge \texttt{ship\_type} = \lambda \\ & \wedge \texttt{coupon} = \lambda \wedge \texttt{amount\_owed} = \lambda \wedge \texttt{amount\_paid} = \lambda \\ & \wedge \texttt{amount\_refunded} = \lambda\end{aligned}$$

The services. The following services model a few of the business process tasks (for a complete list, see Appendix 2.4).

choose_product The customer chooses a product among the available ones.

π : status = "edit_prod"

$\psi : \exists p, a, w(\texttt{PRODUCTS}(\texttt{prod\_id}', p, a, w) \wedge a > 0) \wedge \texttt{status}' =$ "edit_shiptype"

choose_shiptype The customer chooses a shipping option.

π : status = "edit_ship"

ψ : $\exists c, l, p, a, w$(SHIPPING(ship_type$'$, c, l)$\wedge$
PRODUCTS(prod_id, p, a, w) $\wedge$ $l > w$)$\wedge$
status$'$ = "edit_coupon" $\wedge$ prod_id$'$ = prod_id

apply_coupon The customer optionally inputs a coupon number.

π : status = "edit_coupon"
ψ : (coupon$'$ = λ $\wedge$ $\exists p, a, w, c, l$(PRODUCTS(prod_id, p, a, w)
$\wedge$SHIPPING(ship_type, c, l) $\wedge$ amount_owed$'$ = $p + c$)$\wedge$
status$'$ = "processing" $\wedge$ prod_id$'$ = prod_id$\wedge$
ship_type$'$ = ship_type)$\vee$
($\exists t, v, m, s, p, a, w, c, l$(COUPONS(coupon$'$, t, v, m, s)$\wedge$
PRODUCTS(prod_id, p, a, w) $\wedge$ SHIPPING(ship_type, c, l)
$\wedge p + c \geq m \wedge$ (t = "free_shipping" $\rightarrow$
(s = ship_type $\wedge$ amount_owed$'$ = p))$\wedge$
(t = "discount" $\rightarrow$ amount_owed$'$ = $p + c - v$))
$\wedge$status$'$ = "processing" $\wedge$ prod_id$'$ = prod_id$\wedge$
ship_type$'$ = ship_type)

edit_product The customer chooses another product.

π :status = "edit_shiptype" $\vee$ status = "edit_coupon" $\vee$
status = "processing"
ψ : status$'$ = "edit_prod"

edit_shiptype The customer chooses another shipment type.

π : status = "edit_coupon" $\vee$ status = "processing"
ψ : status$'$ = "edit_shiptype" $\wedge$ prod_id$'$ = prod_id

edit_coupon The customer applies a different coupon.

π : status = "processing"
ψ :status$'$ = "edit_coupon" $\wedge$ prod_id$'$ = prod_id$\wedge$
ship_type$'$ = ship_type

receive_payment The customer sends a payment.

$$\pi : \texttt{status} = \text{“processing”}$$
$$\psi : \texttt{amount\_paid}' \geq 0 \wedge \texttt{status}' = \text{“received\_payment”} \wedge$$
$$\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge$$
$$\texttt{coupon}' = \texttt{coupon} \wedge \texttt{amount\_owed}' = \texttt{amount\_owed}$$

check_payment Check payment and ship the product.

$$\pi : \texttt{status} = \text{“received\_payment”}$$
$$\psi : ((\texttt{amount\_paid} \neq \texttt{amount\_owed}) \rightarrow$$
$$(\texttt{status}' = \text{“received\_payment”} \wedge \texttt{amount\_paid}' = 0)) \wedge$$
$$((\texttt{amount\_paid} = \texttt{amount\_owed}) \rightarrow$$
$$(\texttt{status}' = \text{“shipping”} \wedge \texttt{amount\_paid}' = \texttt{amount\_paid})) \wedge$$
$$\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge$$
$$\texttt{coupon}' = \texttt{coupon} \wedge \texttt{amount\_owed}' = \texttt{amount\_owed}$$

ship A paid order gets shipped.

$$\pi : \texttt{status} = \text{“shipping”}$$
$$\psi : \texttt{status}' = \text{“shipped”} \wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge$$
$$\texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{coupon}' = \texttt{coupon} \wedge$$
$$\texttt{amount\_owed}' = \texttt{amount\_owed} \wedge$$
$$\texttt{amount\_paid}' = \texttt{amount\_paid}$$

cancel_order The customer cancels the order.

$$\pi : \neg(\texttt{status} = \text{“canceled”} \vee \texttt{status} = \text{“canceling”})$$
$$\psi : (\texttt{amount\_paid} = 0 \rightarrow \texttt{status}' = \text{“canceled”}) \wedge$$
$$(\texttt{amount\_paid} > 0 \rightarrow \texttt{status}' = \text{“canceling”}) \wedge$$
$$\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge$$
$$\texttt{coupon}' = \texttt{coupon} \wedge \texttt{amount\_owed}' = \texttt{amount\_owed}$$
$$\wedge \texttt{amount\_paid}' = \texttt{amount\_paid}$$

refund_payment Payment is refunded in case of canceled order.

$$\pi : \texttt{status} = \text{“canceling”}$$
$$\psi : \texttt{amount\_paid}' = 0$$
$$\wedge \texttt{amount\_refunded}' = \texttt{amount\_paid} \wedge$$
$$\texttt{status}' = \text{“canceled”} \wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge$$
$$\texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{coupon}' = \texttt{coupon} \wedge$$
$$\texttt{amount\_owed}' = \texttt{amount\_owed}$$

Notice that the pre-conditions π of the services check the value of the `status` variable. For instance, according to **choose_product**, the customer can only input her product choice while the order is in “edit_prod” status.

Also notice that the post-conditions ψ set the next values of the artifact variables (denoted by a prime). For instance, according to **choose_product**, once a product has been picked, the next value of the status variable is “edit_shiptype”, which will at a subequent step enable the **choose_shiptype** service (by satisfying its pre-condition). Similarly, once the shipment type is chosen (as modeled by service **choose_shiptype**), the new status is “edit_coupon”, which enables the **apply_coupon** service.

A post-condition may refer to both the current and next values of the artifact variables. For instance, in service **choose_shiptype**, the fact that only the shipment type is picked, while the product is unchanged, is modeled by preserving the product id: the next and current values of the corresponding artifact variable are set equal.

Pre- and post-conditions may query the database. For instance, in service **choose_product**, the post-condition ensures that the product id chosen by the customer is that of an available product (by checking that it appears in a `PRODUCTS` tuple, whose availability attribute is positive).

Finally, notice the arithmetic computation in the post-conditions. For instance, in service **apply_coupon**, the sum of the product price p and shipment cost c (looked up in the database) is adjusted with the coupon value (notice the distinct treatment of the two coupon types) and stored in the `amount_owed` artifact variable.

We also assume some status values as *final*, meaning that the order can rest in those states forever. These states are “shipped” and “canceled”. This is modeled by two dummy services that simply repeat any configuration with this status:

shipped_dummy A shipped order remains shipped.

$$\pi : \texttt{status} = \text{“shipped”}$$
$$\psi : \texttt{status}' = \texttt{status} \wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge$$
$$\texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{coupon}' = \texttt{coupon} \wedge$$
$$\texttt{amount\_owed}' = \texttt{amount\_owed}$$
$$\wedge \texttt{amount\_paid}' = \texttt{amount\_paid}$$
$$\wedge \texttt{amount\_refunded}' = \texttt{amount\_refunded}$$

canceled_dummy A canceled order remains canceled.

$$\pi : \texttt{status} = \text{“canceled”}$$
$$\psi : \texttt{status}' = \texttt{status} \wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge$$
$$\texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{coupon}' = \texttt{coupon} \wedge$$
$$\texttt{amount\_owed}' = \texttt{amount\_owed}$$
$$\wedge \texttt{amount\_paid}' = \texttt{amount\_paid}$$
$$\wedge \texttt{amount\_refunded}' = \texttt{amount\_refunded}$$

Some properties

We show a few properties that specify desirable business rules for the running example.

$$(\varphi_1)\ \forall x \mathbf{G}((\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}) \rightarrow \mathbf{F}(\texttt{status} = \text{”shipped”} \vee \texttt{amount\_refunded} = x))$$

Property φ_1 states that if a correct payment is submitted then at some time in the future either the product is shipped or the customer is refunded the correct amount. φ_1 is obtained from LTL property $\varphi = \mathbf{G}(p \rightarrow \mathbf{F}q)$ via the mapping f_1, where $f_1(p) = \texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}$ and $f_1(q) = \texttt{status} = \text{”shipped”} \vee \texttt{amount\_refunded} = x$. Note the use of universally-quantified variable x to relate the value of paid and refunded amounts across distinct steps in the run sequence.

$$(\varphi_2)\ \forall\, v,m,s,p,a,w,c,l(\mathbf{G}(\texttt{prod\_id} \neq \lambda \wedge \texttt{ship\_type} \neq \lambda$$

$\wedge$COUPONS(coupon,"free_ship",v,m,s))$\wedge$
PRODUCTS(prod_id,p,a,w)$\wedge$SHIPPING(ship_type,c,l)
$\rightarrow \underbrace{a>0}_{(i)} \wedge \underbrace{w \leq l}_{(ii)} \wedge \underbrace{p+c \geq m}_{(iii)})$

Property φ_2 verifies the consistency of orders that use coupons for free shipping. The premise of the implication lists the conditions for a completely specified order that uses such coupons. The conclusion checks the following business rules (i) available quantity of the product is greater than zero, (ii) the weight of the product is in the limit allowed by the shipment method, and (iii) the total order value satifies the minimum for the application of the coupon.

Note that this property holds only due to the integrity constraints on the schema. Indeed, observe that (i) is guaranteed by the post-condition of service **choose_product**, (ii) by **choose_shiptype**, and (iii) by **apply_coupon**. In the post-conditions, the checks are perfomed by looking up in the database the weight/price/cost/limit attributes associated to the customer's selection of product id and shipment type (stored in artifact variables). The property performs the same lookup in the database, and it is guaranteed to retrieve the same tuples only because product id and shipment type are keys for PRODUCTS, respectively SHIPPING. The verifier must take these key declarations into account, to avoid generating a spurious counter-example in which the tuples retrieved by the service post-conditions are distinct from those retrieved by the property, despite agreeing on product id and shipment type.

(φ_3) $\forall p,a($**G**(OFFERS(prod_id,p,a) $\rightarrow$
F(status = "shipped" $\vee$ status = "canceled"))).

(φ_3) checks that for products offered on sale, the business process eventually reaches either the "shipped" or "canceled" status. Notice that the appropriate services manipulate explicitly only the PRODUCTS table, so a property involving the OFFER table would have no reason to hold in general. It does hold in particular, due to the foreign key between OFFERS and PRODUCTS.

Symbolic transition templates and computation graph

In Example 2.3.1 and Figure 6.1, we illustrate symbolic transition templates and the associated graphs $G_\psi, E_\psi, G_\rho, E_\rho$ for the property

$$\varphi_f = \mathbf{F}(\texttt{status} = \text{“shipped”} \vee \texttt{status} = \text{“canceled”}).$$

We provide here further details for this example.

To build Δ_Σ, we need to rewrite the conjunction of pre- and post-condition for each service into prenex DNF, and add each disjunct as a separate formula to Δ_Σ. The pre- and post-conditions of services **choose_product** and **choose_shiptype** are conjunctive, so only trivial prenex normal form rewriting is needed, which we omit. For service **apply_coupon**, we obtain five disjuncts $\bigvee_{i=1}^{5} \xi_i$ for the DNF of $\pi \wedge \psi$:

ξ_1: $\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{coupon}' = \lambda \wedge \texttt{status} =$
“edit_coupon” $\wedge\, \texttt{status}' =$ “processing”$\wedge$
$\exists p,a,w,c,l(\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w)\wedge$
$\texttt{SHIPPING}(\texttt{ship\_type},c,l) \wedge \texttt{amount\_owed}' = p+c)$

ξ_2: $\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{status} =$ “edit_coupon” $\wedge$
$\texttt{status}' =$ “processing”$\wedge$
$\exists t,v,m,s,p,a,w,c,l(\texttt{COUPONS}(\texttt{coupon}',t,v,m,s)\wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge \texttt{SHIPPING}(\texttt{ship\_type},c,l) \wedge p+c \geq m \wedge \neg(t =$
“free_shipping”$) \wedge \texttt{amount\_owed}' = p+c-v)$

ξ_3: $\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{status} =$ “edit_coupon” $\wedge$
$\texttt{status}' =$ “processing”$\wedge$
$\exists t,v,m,s,p,a,w,c,l(\texttt{COUPONS}(\texttt{coupon}',t,v,m,s)\wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge \texttt{SHIPPING}(\texttt{ship\_type},c,l) \wedge p+c \geq m \wedge \neg(t =$
“discount”$) \wedge s = \texttt{ship\_type} \wedge \texttt{amount\_owed}' = p)$

ξ_4: $\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{status} =$ “edit_coupon” $\wedge$
$\texttt{status}' =$ “processing”$\wedge$
$\exists t,v,m,s,p,a,w,c,l(\texttt{COUPONS}(\texttt{coupon}',t,v,m,s)\wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id},p,a,w) \wedge \texttt{SHIPPING}(\texttt{ship\_type},c,l) \wedge p+c \geq m \wedge s =$
$\texttt{ship\_type} \wedge \texttt{amount\_owed}' = p \wedge \texttt{amount\_owed}' = p+c-v)$

ξ_5: $\texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type} \wedge \texttt{status} =$ "edit_coupon" $\wedge$
$\texttt{status}' =$ "processing" $\wedge$
$\exists t, v, m, s, p, a, w, c, l(\texttt{COUPONS}(\texttt{coupon}', t, v, m, s) \wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge \texttt{SHIPPING}(\texttt{ship\_type}, c, l) \wedge p + c \geq m \wedge \neg(t =$
"free_shipping") $\wedge \neg(t =$ "discount"))

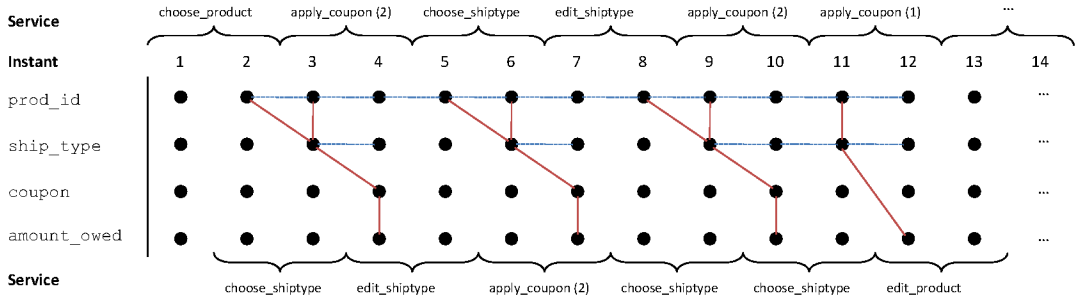

Figure 2.3: A computation graph for running example

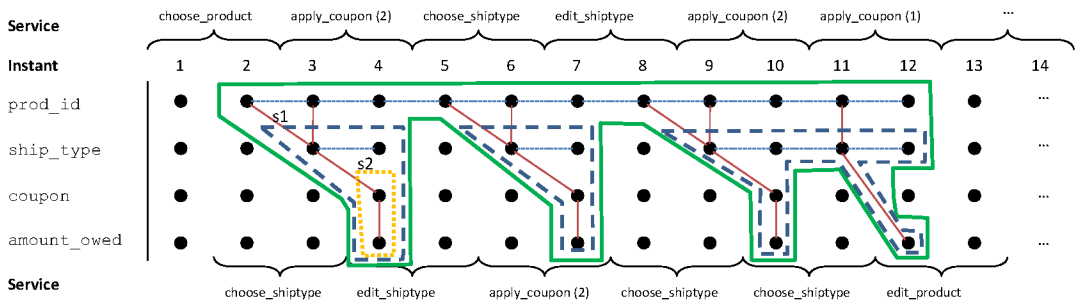

Figure 2.4: Structure of computation graph from Figure 2.3 induced by connnected components and their nested sub-components

Inherited Constraints

We illustrate the construction of inherited constraints on the running example. Consider a symbolic run $\rho = \{\psi_i\}_{i>0}$, whose computation graph G_ρ is represented in Figure 2.3.

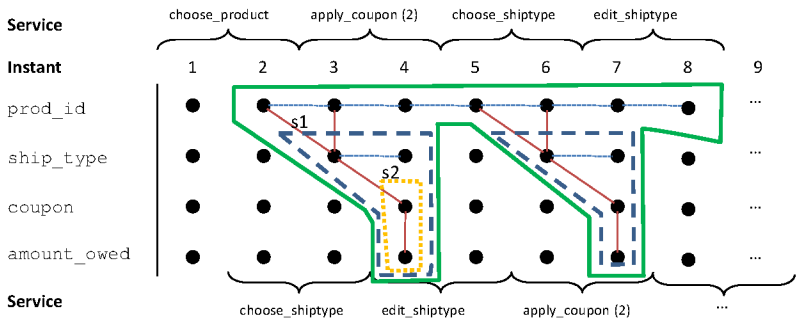

Figure 2.5: Prefix of graph from Figure 2.4 relevant to the inherited constraint at step 8

Before showing the examples of the inherited constraints, we name certain subformulas in order to increase readability.

$$\chi_1(prod\_id) = \exists p,a,w(\texttt{PRODUCTS}(prod\_id,p,a,w) \wedge a > 0)$$

$$\chi_2(prod\_id, ship\_type) = \exists c,l,p,a,w(\texttt{SHIPPING}(ship\_type,c,l)$$
$$\wedge \texttt{PRODUCTS}(prod\_id,p,a,w) \wedge l > w)$$

$$\chi_3(prod\_id, ship\_type, coupon, amount\_owed) =$$
$$\exists t,v,m,s,p,a,w,c,l(\texttt{COUPONS}(coupon,t,v,m,s) \wedge$$
$$\texttt{PRODUCTS}(prod\_id,p,a,w) \wedge \texttt{SHIPPING}(ship\_type,c,l) \wedge$$
$$p + c \geq m \wedge \neg(t = \text{“free\_shipping”}) \wedge$$
$$amount\_owed = p + c - v)$$

We consider now the inherited constraint at instant 5,

$$\eta_5(\bar{x}_5) = \exists \bar{z}(\psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4),$$

where ψ_i are shown below, $\bar{x}_5 = \{\texttt{prod\_id}_5, \texttt{ship\_type}_5, \texttt{coupon}_5, \texttt{amount\_owed}_5\}$, and the quantified variables $\bar{z}$ include the 7 other artifact variables mentioned in ψ_2 through ψ_4, as well as the 3+5+9 variables in χ_1, χ_2, χ_3 respectively. Prior to rewriting, the quantifier rank of η_5 is 24.

We show below the symbolic transition templates ψ_i instantiated with the the symbolic run variables:

$$\psi_1 : \texttt{status}_1 = \text{“edit\_prod”} \wedge \chi_1(\texttt{prod\_id}_2) \wedge$$

$\texttt{status}_2 =$ "edit_shiptype"

$\psi_2 : \texttt{status}_2 =$ "edit_ship" $\wedge\, \chi_2(\texttt{prod\_id}_2, \texttt{ship\_type}_3) \wedge$
$\texttt{status}_3 =$ "edit_coupon" $\wedge\, \texttt{prod\_id}_3 = \texttt{prod\_id}_2$

$\psi_3 : \texttt{status}_3 =$ "edit_coupon" $\wedge\, \texttt{status}_4 =$ "processing" $\wedge$
$\texttt{prod\_id}_4 = \texttt{prod\_id}_3 \wedge \texttt{ship\_type}_4 = \texttt{ship\_type}_3 \wedge$
$\chi_3(\texttt{prod\_id}_3, \texttt{ship\_type}_3, \texttt{coupon}_4, \texttt{amount\_owed}_4)$

$\psi_4 : \texttt{status}_4 =$ "processing" $\wedge\, \texttt{status}_5 =$ "edit_shiptype" $\wedge$
$\texttt{prod\_id}_5 = \texttt{prod\_id}_4$

We illustrate η_5^* next, which (recall from Section 3.1) is the rewriting of η_5 by pushing quantifiers inside. To simplify presentation we abstract away from the `status` artifact variable.

$$\eta_5^* = \chi_1([\texttt{prod\_id}_2]) \wedge \exists[\texttt{ship\_type}_3](\chi_2([\texttt{prod\_id}_2], [\texttt{ship\_type}_3]) \wedge \exists[\texttt{coupon}_4], [\texttt{amount\_owed}_4](\chi_3([\texttt{prod\_id}_2], [\texttt{ship\_type}_3], [\texttt{coupon}_4], [\texttt{amount\_owed}_4])))$$

The forest representation of η_5^* is [2] :

$$\eta_5^* = [\texttt{prod\_id}_2](\chi_1([\texttt{prod\_id}_2]), [\texttt{ship\_type}_3](\chi_2([\texttt{prod\_id}_2], [\texttt{ship\_type}_3]), [\texttt{coupon}_4]([\texttt{amount\_owed}_4](\chi_3([\texttt{prod\_id}_2], [\texttt{ship\_type}_3], [\texttt{coupon}_4], [\texttt{amount\_owed}_4])))))$$

Notice the dramatic reduction in quantifier rank, from 24 to 11. The quantifier rank of η_5^* is 11, since sub-formula χ_3 has quantifier rank 9 (maximum among χ_1, χ_2, χ_3), and it occurs in the scope of the two quantified variables $[\texttt{coupon}_4], [\texttt{amount\_owed}_4]$.

To follow how η_5^* was constructed, recall that the construction recursively splits the computation graph into connected components, obtaining the formula for each component from those of its sub-components. For instance, notice that the sub-component

[2] Convention: we slightly modify the simplified syntax tree notation for ∃FO formulas from Section **??**, to avoid a clash between the angular brackets used for variable equivalence classes and the ones used for the children of a node. We show the list of children in parentheses instead.

labeled $s1$ in Figure 2.4 is obtained after removing the maximum-span equivalence class $[\texttt{prod\_id}_2]$. In turn, nested sub-component s_2 is obtained after removing from s_1 its maximum-span equivalence class
$[\texttt{ship\_type}_3]$.

The nested sub-component structure mirrors the forest representation of the resulting formula. To illustrate, notice that for η_5^*, the sub-formula μ_1 corresponds to component s_1, and μ_2 to s_2:

$$\mu_1 = \exists[\texttt{ship\_type}_3](\chi_2([\texttt{prod\_id}_2],[\texttt{ship\_type}_3]) \wedge \exists [\texttt{coupon}_4],[\texttt{amount\_owed}_4](\chi_3([\texttt{prod\_id}_2],[\texttt{ship\_type}_3], [\texttt{coupon}_4],[\texttt{amount\_owed}_4])))$$

$$\mu_2 = \exists[\texttt{coupon}_4],[\texttt{amount\_owed}_4](\chi_3([\texttt{prod\_id}_2],[\texttt{ship\_type}_3], [\texttt{coupon}_4],[\texttt{amount\_owed}_4]))$$

Intuitively, μ_1 relates the product id chosen by the user at step 2 ($[\texttt{prod\_id}_2]$) with the shipment type picked at step 3 ($[\texttt{ship\_type}_3]$), then μ_2 relates the latter with the coupon claimed at step 4
($[\texttt{coupon}_4]$).

Finally, we illustrate reduced inherited constaints, and how they repeat during the run. Consider the inherited constraint at step 8, η_8^*, derived from the computational graph prefix in Figure 2.5. It is easy to see that the forest representation of η_8^* has two identical children of the root node $[\texttt{prod\_id}_2]$, corresponding to the sub-components delineated by blue dashed edges:

$$\eta_8^* = [\texttt{prod\_id}_2](\chi_1([\texttt{prod\_id}_2]),\mu_1([\texttt{prod\_id}_2]), \chi_1([\texttt{prod\_id}_2]),\mu_1([\texttt{prod\_id}_2]))$$

Observe that the reduced constraints coincide:

$$red(\eta_5^*) = red(\eta_8^*) = \eta_5^*.$$

Acknowledgement

Alin Deutsch and Victor Vianu co-authored this chapter.

3 Verification of Feedback-free Systems

The main result of this section is that model checking LTL-FO properties is decidable if the artifact system together with the property are feedback-free. In Section 3.2, we extend this result to the presence of integrity constraints on database relations.

Theorem 3.0.1. *It is decidable, given an artifact system* Γ *and an LTL-FO formula* $\forall \bar{y} \varphi_f$ *such that* $(\Gamma, \forall \bar{y} \varphi_f)$ *is feedback-free, whether* $\forall \bar{y} \varphi_f$ *holds for every run* ρ *of* Γ.

The proof requires developing some technical machinery, and is outlined in the remainder of the chapter. In the next section we consider the case with no data dependencies, and we will later show how to take them into account.

3.1 Verification with only arithmetic

Let $\Gamma = \langle \mathscr{A}, \Sigma, \Pi \rangle$ where $\mathscr{A} = \langle \bar{x}, \mathscr{D}\mathscr{B} \rangle$. Recall that, by Lemma 2.2.3, one can eliminate the global variables $\bar{y}$ of the LTL-FO formula $\forall \bar{y} \varphi_f$. Thus, we can assume the LTL-FO formula to be verified is simply φ_f. Clearly, $\Gamma \models \varphi_f$ iff there is no run of Γ that satisfies $\neg \varphi_f$. We will prove the theorem by showing decidability of the latter property.

The verification algorithm makes use of the symbolic runs introduced earlier. We claim that symbolic runs provide a representation of all actual runs of an artifact system. Let $\rho = \{\psi_i\}_{i \geq 0}$ be a symbolic run of Γ. To each such symbolic run and each database instance D we associate a set of actual runs on D as follows. Let $var(\rho) = \{\bar{x}_i \mid i \geq 0\}$ and $\Delta_\rho = \{\psi_i | i \geq 0\}$. Note that the set of free variables of formulas in Δ_ρ is $var(\rho)$. Let

$Runs_D(\rho) = \{\{\nu(\bar{x}_i)\}_{i \geq 0} \mid \nu$ is a valuation of $var(\rho)$ into $\mathbb{Q}$

such that $D \cup \mathscr{C} \models \Delta_\rho\}$

We say that ρ is *satisfiable* if there exists a database instance D such that $Runs_D(\rho) \neq \emptyset$. We use analogous defnitions and notation for the case when ρ is a *prefix* of a symbolic run. In particular, for $j > 0$, we denote by $\rho|_j$ the prefix $\{\psi_i\}_{i<j}$ of ρ and refer to $Runs_D(\rho|_j)$ with the obvious meaning.

The following establishes the desired connection between symbolic runs and actual runs.

Lemma 3.1.1. *(i) For each database instance D, $Runs_D(\Gamma) = \cup\{Runs_D(\rho) \mid \rho$ is a symbolic run of $\Gamma\}$. (ii) There exists a run of Γ satisfying $\neg\varphi_f$ iff there exists a satisfiable symbolic run of Γ satisfying $\neg\varphi_f$.*

Proof: Consider (i). Suppose $\rho = \{\rho_i\}_{i\geq 0}$ is a run of Γ on database D. By definition, for each $i \geq 0$ there exists a formula $\delta(\bar{x}_i, \bar{x}_{i+1}) \in \Delta_\Sigma$ so that $D \cup \mathscr{C} \models \delta(\rho_i, \rho_{i+1})$. Also, for each FO component $f(p)$ of $\neg\varphi_f$, either $D \cup \mathscr{C} \models f(p)(\rho_i, \rho_{i+1})$ or $D \cup \mathscr{C} \models \neg f(p)(\rho_i, \rho_{i+1})$. It follows that there exists a formula ψ_i constructed as above from Δ_Σ and Δ_{φ_f} so that $D \cup \mathscr{C} \models \psi_i(\rho_i, \rho_{i+1})$. (For ψ_0, the pre-condition Π is also taken into account in the obvious way.) Let $\rho = \{\psi_i\}_{i\geq 0}$. By construction, $\rho \in Runs_D(\rho)$. The converse is similar.

Part (ii) follows from the observation that for every run $\rho \in Runs_D(\rho)$, the same FO components of φ_f are satisfied in the i-th configuration of ρ and ρ. Thus, $\rho \models \neg\varphi_f$ iff $\rho \models \neg\varphi_f$. □

In view of the above, it is sufficient to check the existence of a satisfiable symbolic run of Γ satisfying $\neg\varphi_f$. To prove decidability, we show that it is enough to consider prefixes of symbolic runs of statically bounded length.

Consider a symbolic run $\{\psi_i\}_{i\geq 0}$. We define for each $j > 0$ the *inherited constraint* of configuration j, denoted η_j, which summarizes the constraints on configuration j imposed by the prefix leading to it, i.e. $\{\psi_i\}_{i<j}$. The inherited constraint $\eta_j(\bar{x}_j)$ is defined as $\exists\bar{z}\bigwedge_{i<j}\psi_i$ where $\bar{z}$ are all variables other than $\bar{x}_j$. The following is immediate from the definitions.

Lemma 3.1.2. *Let $\rho = \{\psi_i\}_{i\geq 0}$ be a symbolic run and D a database instance. Then for every $j > 0$,*

$$\{\rho_j \mid \{\rho_i\}_{0\leq i\leq j} \in \mathit{Runs}_D(\rho|_j)\} = \{\rho_j \mid D \cup \mathscr{C} \models \eta_j(\rho_j)\}.$$

In other words, η_j defines precisely the set of valuations of $\bar{x}$ reachable in the last configuration of a run in $\mathit{Runs}_D(\rho|_j)$. Note that this is generally a strict superset of $\{\rho_j \mid \{\rho_i\}_{i\geq 0} \in \mathit{Runs}_D(\rho)\}$.

We next show the key fact that for a feedback-free system there are only finitely many inherited constraints up to logical equivalence. This is done by rewriting each such constraint as a $\mathrm{CQ}^{\neg}$ formula of bounded quantifier rank. Recall that the quantifier rank of a formula is the maximum number of quantifiers along a path from root to leaf in the syntax tree of the formula, and that there are finitely many non-equivalent formulas of given quantifier rank over a given vocabulary (e.g., see [Lib04]). The number of non-equivalent formulas is hyperexponential in the quantifier rank.

Lemma 3.1.3. *For each η_j one can construct an equivalent $CQ^{\neg}$ formula $\bar{\eta}_j$ of quantifier rank bounded by $k^2 + q$, where $k = |\bar{x}|$ and q is the maximum quantifier rank of the formulas used in pre-and-post conditions of services in Σ.*

The coming proof shows how to rewrite η_j as a formula η_j^* of quantifier rank at most $k^2 + q$, whose variables are the equivalence classes of variables in $\rho|_j$ induced by the equality graph $E^*_{\rho|_j}$. The construction of η_j^* is non-deterministic, so several outcomes are possible, all with the same quantifier rank. Finally, $\bar{\eta}_j$ is obtained from η_j^* by replacing its free variables with $\bar{x}_j$. We will refer to η_j^* in the sequel.

Proof: Let η_j be defined as above. Recall that service pre-and-post conditions may contain $\exists$FO formulas, so the ψ_i may generally contain such subformulas. We assume first for simplicity that the ψ_i are quantifier free, and deal with the $\exists$FO subformulas later. We denote by $\xi = \bigwedge_{i<j} \psi_i$, so $\eta_j = \exists \bar{z} \xi$. Consider the equivalence relation induced by the equality atoms in η_j on the variables $\{\bar{x}_i\}_{i\leq j}$. We denote the equivalence class of y by $[y]$ (when η_j is understood). By slight abuse, we use such equivalence classes as variables in the fomula we are constructing. Thus, we begin by eliminating all positive equality atoms from η_j and replacing in each non-equality atom each variable y by its

equivalence class $[y]$. Let us denote by $[\eta_j]$ the resulting formula. Note that the set of free variables of $[\eta_j]$ is now the set of equivalence classes of variables in $\bar{x}_j$, denoted $[\bar{x}_j]$. It is clear by construction that for every database D, $D \cup \mathscr{C} \models \eta_j(\bar{x}_j)$ iff $D \cup \mathscr{C} \models [\eta_j]([\bar{x}_j])$. We will ensure later that the final formula $\bar{\eta}_j$ has free variables $\bar{x}_j$ rather than $[\bar{x}_j]$.

Note that $[\eta_j]$ is in existential prenex form. We next show that the quantifiers can be pushed inside, resulting in a formula η_j^* of quantifier rank bounded by k^2. To this end, we will use the computation graph associated with a symbolic run (or a prefix thereof). Let us denote by G_j the graph on equivalence classes induced by the computation graph of ξ. More precisely, if there is an edge from x_i to y_n in G_ξ, then there is an edge from $[x_i]$ to $[y_n]$ in G_j. Consider a formula β consisting of the conjunction of a subset of the literals of $[\xi]$ and let G_β be its associated computation graph (so a subgraph of G_j). Let $\bar{y}$ be a subset of the variables of β. The *width* of $\bar{y}$ is defined as

$$w(\bar{y}) \;=\; max\{|\bar{v}| \;\mid \bar{v} \subseteq \bar{x}_i, \text{ and each } v \in \bar{v} \text{ is in an equivalence class } y \in \bar{y}\}.$$

We prove the statement for formulas $\gamma(\bar{s}) = \exists \bar{y} \beta(\bar{y}, \bar{s})$ with free variables $\bar{s}$, such that the restriction of G_β to $\bar{y}$, denoted $G_\beta|\bar{y}$, is connected. More precisely, we show that each such formula can be rewritten with quantifier rank bounded by $k \cdot w(\bar{y})$, where $k = |\bar{x}|$. The proof is by induction on $w(\bar{y})$. If $w(\bar{y}) = 0$ then $\bar{y} = \emptyset$ so $qd(\gamma) = 0$. Now suppose the statement holds for $w(\bar{y}) < n$ and consider $\gamma(\bar{s}) = \exists \bar{y} \beta(\bar{s})$ where $w(\bar{y}) = n$ and $G_\beta|\bar{y}$ is connected. Consider the set S of maximal spans of variables in $\bar{y}$ in ρ_j (with respect to inclusion). For each $s \in S$, let y_s be a variable in $\bar{y}$ for which $span(y_s) = s$, and let $\bar{y}_{max} = \{y_s \mid s \in S\}$. We note the following:

(i) There is no variable $x \in \bar{x}$ and $i \neq m$ such that $[x_i] \neq [x_m]$ and $[x_i], [x_m] \in \bar{y}_{max}$.

Indeed, suppose there were such $[x_i], [x_m]$. Since $G_\beta|\bar{y}$ is connected, there is a path from $[x_i]$ to $[x_j]$ in $G_\beta|\bar{y}$. Since $(\Gamma, \forall \bar{y} \varphi_f)$ is feedback free, the path must go through a class $[t] \in \bar{y}$, whose span strictly includes the spans of both $[x_i]$ and $[x_j]$. Since by definition of $\bar{y}_{max}$ the spans of $[x_i]$ and $[x_m]$ are distinct, the inclusion is strict, contradicting the maximality of the spans of $[x_i]$ and $[x_j]$.

Note that, as a consequence of (i):

(ii) The size of $\bar{y}_{max}$ is bounded by k.

Now let $\bar{v} = \bar{y} - \bar{y}_{max}$ and consider $G_\beta|\bar{v}$. Let $H_1, \ldots, H_c$ be its connected components. For each $r \in [1,c]$ let $\bar{y}_r$ be the set of nodes of H_r and let β_r be the formula consisting of the conjunction of the literals in β using only variables in $\bar{s} \cup \bar{y}_{max} \cup \bar{y}_r$. We claim that:

(iii) $\exists \bar{y} \beta$ is equivalent to the formula $\exists \bar{y}_{max}(\bigwedge_{1 \leq r \leq c} \exists \bar{y}_r(\beta_r))$

To see that (iii) holds, note first that each literal of β belongs to some β_r. Also, $\bar{y} = \bar{y}_{max} \cup \bigcup_{1 \leq r \leq c} \bar{y}_r$. Finally, distinct β_r have no variables in common besides those in $\bar{s} \cup \bar{y}_{max}$.

Now consider a formula $\exists \bar{y}_r \beta_r$. The graph $G_{\beta_r}|\bar{y}_r$ equals H_r so is connected by construction. Consider $w(\bar{y}_r)$. Since $\bar{y}_{max}$ contains variables covering all maximal spans, for each i such that $x \in \bar{x}$ and $[x_i] \in \bar{v}_r$, there exists $v \in \bar{x}$, $v \neq x$, such that $[v_i] \in \bar{y}_{max}$. It follows that $w(\bar{y}_r) \leq w(\bar{y}) - 1$. Thus, we can apply the induction hypothesis and $\exists \bar{y}_r \beta_r$ can be rewritten with quantifier rank at most $k \cdot w(\bar{y}_r)$. It follows that $\exists \bar{y} \beta$ can be rewritten with quantifier rank at most $|\bar{y}_{max}| + k \cdot max\{w(\bar{y}_r) \mid 1 \leq r \leq c\}$. Since by (ii) $|\bar{y}_{max}| \leq k$, this is bounded by $k(1 + (w(\bar{y}) - 1) = k \cdot w(\bar{y})$.

In summary, we have shown that every formula $\exists \bar{y} \beta$ where $G_\beta|\bar{y}$ is connected can be rewritten with quantifier rank at most $k \cdot w(\bar{y})$. Finally, consider $[\eta_j] = \exists[\bar{z}][\xi]$. Similarly to the induction step, consider the connected components $H_1, \ldots, H_c$ of $G_j|[\bar{z}]$. For each $r \in [1,c]$ let $\bar{y}_r$ be the set of nodes of H_r and let ξ_r be the formula consisting of the conjunction of the literals in ξ using only variables in $[\bar{x}_j] \cup \bar{y}_r$. It is clear that $[\eta_j]$ is equivalent to the formula $\bigwedge_{1 \leq r \leq c} \exists \bar{y}_r(\xi_r)$. By construction, the graph $G_{\xi_r}|\bar{y}_r$ is connected for each r, so by the above $\exists \bar{y}_r(\xi_r)$ can be rewritten with quantifier rank bounded by $k \cdot w(\bar{y}_r) \leq k^2$ for each r. It follows that $[\eta_j]$ can be rewritten with the same quantifier rank k^2.

Recall that we have assumed that all formulas ψ_i are quantifier free. If $\exists$FO formulas are used in Γ, the above construction can be extended as follows. First, associate to each maximal subformula α used in Γ such that $\alpha(\bar{u}) = \exists \bar{v} \gamma(\bar{v}, \bar{u})$, where $\bar{u}$ are the free variables, a new relation symbol R_α of arity $|\bar{u}|$, and replace in Γ each occurrence

of $\alpha(\bar{u})$ by $R_\alpha(\bar{u}) \wedge eq(\bar{u})$, where $eq(\bar{u})$ is the conjunction of all equalities among variables in $\bar{u}$, resulting from taking the transitive closure of the equality graph of $\gamma(\bar{v},\bar{u})$. The above construction then yields a $\mathrm{CQ}^\neg$ formula of quantifier rank k^2 over the augmented vocabulary. Finally, replace each atom $R_\alpha(\bar{u})$ by $\alpha(\bar{u})$, yielding a $\mathrm{CQ}^\neg$ formula of quantifier rank k^2+q.

Finally, to obtain a formula equivalent to η_j, we define $\bar{\eta}_j$ from η_j^* as follows. For each equivalence class $e \in [\bar{x}_j]$ let v_e be an arbitrarily chosen variable in $\bar{x}_j$ such that $v_e \in e$. Let $\bar{\eta}(\bar{x}_j)$ be obtained by substituting v_e for each e in $\eta^*([\bar{x}_j])$ and adding the conjunction of all equalities $\{u = v \mid u,v \in \bar{x}_j, [u] = [v]\}$. The quantifier rank of $\bar{\eta}_j$ remains k^2+q and $\bar{\eta}_j$ is equivalent to η_j. □

3.1.1 Reduced inherited constraints

It is well known that the number of formulas of given quantifier rank is finite, up to logical equivalence. The notion of equivalence is in fact a strong syntactic one, upon which we elaborate next. For every $\mathrm{CQ}^\neg$ formula α, we can define a reduction procedure yielding a logically equivalent formula $red(\alpha)$, obtained essentially by recursively merging isomorphic subformulas. It can be seen that the number of distinct reduced formulas of given quantifier rank d is bounded by a hyperexponential in d. This also yields our upper bound for inherited constraints.

We elaborate briefly on the procedure for constructing $red(\alpha)$ for a $\mathrm{CQ}^\neg$ formula α. For each such α we first define a simplified representation of its syntax tree as a forest $\mathfrak{I}(\alpha)$ as follows (a tree consisting of root r and child subtrees $t_1,\ldots,t_n$ is denoted $r[t_1,\ldots,t_n]$):

- $\mathfrak{I}(L) = \{L\}$ for a literal L
- $\mathfrak{I}(\beta_1 \wedge \beta_2) = \mathfrak{I}(\beta_1) \cup \mathfrak{I}(\beta_2)$
- $\mathfrak{I}(\exists z(\beta)) = \{z[\mathfrak{I}(\beta)]\}$

Thus, all internal nodes of $\mathfrak{I}(\alpha)$ are labeled by variables, and the leaves are literals. We define by structural recursion a partial order $\prec$ and then an equivalence relation $\sim$ on $\mathrm{CQ}^\neg$ formulas that have the same free variables as follows. First, $\prec$ is defined on trees as follows:

- for literals L_1, L_2, $L_1 \prec L_2$ iff $L_1 = L_2$;
- for trees $z_1[F_1], z_2[F_2]$ (where z_1, z_2 are variables and F_1, F_2 forests) let z be a new variable. Then $z_1[F_1] \prec z_2[F_2]$ iff for each $t_1 \in F_1$ there exists $t_2 \in F_2$ such that $t_1(z_1 \leftarrow z) \prec t_2(z_2 \leftarrow z)$.

For CQ$^\neg$ formulas φ_1 and φ_2 with the same free variables, $\varphi_1 \prec \varphi_2$ if for each $t_1 \in \mathfrak{T}(\varphi_1)$ there exists $t_2 \in \mathfrak{T}(\varphi_2)$ such that $t_1 \prec t_2$. Finally, $\varphi_1 \sim \varphi_2$ iff $\varphi_1 \prec \varphi_2$ and $\varphi_2 \prec \varphi_1$.

Example 3.1.4 The forest corresponding to

$$Q(y) \wedge \exists x(R(x,y) \wedge P(y) \wedge \exists z(R(y,z) \wedge \neg R(z,z)) \\ \wedge \exists u(R(y,u) \wedge \neg R(u,u)))$$

is

$$[Q(y), x[R(x,y), P(y), z[R(y,z), \neg R(z,z)], \\ u[R(y,u), \neg R(u,u)]]]$$

□

We note the following property, immediate from the definition.

Lemma 3.1.5. *If $\varphi_1 \sim \varphi_2$ then φ_1 and φ_2 are equivalent.*

Using the above, we define a normal form for CQ$^\neg$ formulas as follows. For each α, the *reduced* formula $red(\alpha)$ is obtained by eliminating multiple equivalent sibling subtrees, i.e. retaining only one representative of each equivalence class of $\sim$ among all sibling subtrees.

Example 3.1.6 In the forest of Example 3.1.4, the subtrees rooted at variables z and u are $\sim$-equivalent, and the reduced formula has the forest $[Q(y), x[R(x,y), P(y), z[R(y,z), \neg R(z,z)]]]$. □

We observe the following.

Lemma 3.1.7. *For CQ$^\neg$ formulas φ_1, φ_2, $\varphi_1 \sim \varphi_2$ iff $red(\varphi_1) = red(\varphi_2)$.*

In particular, $\alpha \sim red(\alpha)$ and α is equivalent to $red(\alpha)$.

We denote by Hyp the class of hyperexponential functions. Each function in Hyp is defined inductively by $hyp(0) = 1$ and $hyp(n+1) = 2^{c \cdot hyp(n)}$ for some constant c.

Lemma 3.1.8. *Let Γ and $\forall \bar{y} \varphi_f$ be as above. The number of distinct reduced inherited constraints in configurations of symbolic runs is bounded by $h(k^2)$ for some $h \in Hyp$.*

Proof: By Lemma 2.2.3 it follows that there exists Γ' with $k + |\bar{y}|$ artifact variables such that $\Gamma \models \forall \bar{y} \varphi_f$ iff $\Gamma' \models \varphi_f$. However, the variables in $\bar{y}$ are used in Γ' in a very limited way, and do not contribute to the quantifier rank of inherited constraints. Indeed, in every inherited constraint η_j there is a unique equivalence class for every $y \in \bar{y}$, which occurs free in η_j. The construction in Lemma 3.1.3 then yields a rewriting η_j^* of η_j with quantifier rank k^2 over a fixed vocabulary consisting of the relations in $\mathscr{DB}$ and $\mathscr{C}$ used in Γ and φ_f, as well as the new relations R_α for maximal formulas α used in Γ that have existential quantifications (see above). This in turn yields the hyperexponential bound in k^2. □

Symbolic lassos We next use the finiteness of the reduced inherited constraints to characterize the existence of symbolic runs satisfying $\neg \varphi_f$ using finite prefixes of a certain form, which we call *symbolic lassos*. Let $B_{\neg \varphi}$ be the Büchi automaton corresponding to $\neg \varphi$. Recall that, for a symbolic run $\{\psi_i\}_{i \geq 0}$, we denote by $\{\sigma_i\}_{i \geq 0}$ the sequence of truth assignments to propositions P in φ such that $\sigma_i(p)$ holds iff $\varphi_i \models f(p)$. A run of $B_{\neg \varphi}$ on $\{\sigma_i\}_{i \geq 0}$ is a sequence $\{q_i\}_{i \geq 0}$ of states of $B_{\neg \varphi}$ such that $(init, \sigma_0, q_0)$ is a transition of $B_{\neg \varphi}$ for some initial state *init* and $(q_i, \sigma_{i+1}, q_{i+1})$ is a transition of $B_{\neg \varphi}$ for each $i \geq 0$. A similar definition of run applies to finite sequences $\{\sigma_i\}_{i \leq l}$.

Definition 3.1.9. *A* symbolic lasso *is a finite prefix $\{\psi_i\}_{i < j+n}$ of a symbolic run such that:*

- *$red(\eta_j^*) = red(\eta_{j+n}^*)$,*
- *for each $u, v \in \bar{x}$, $[u_j] = [v_j]$ iff $[u_{j+n}] = [v_{j+n}]$,*
- *for each $u \in \bar{x}$, $[u_j] = [u_{j+n}]$ or $[u_j] \neq [v_{j+n}]$ for each $v \in \bar{x}$,*

- *there exists a run* $\{q_i\}_{i\leq j+n}$ *of* $B_{\neg\varphi}$ *on* $\{\sigma_i\}_{i\leq j+n}$ *such that for some accepting state* r, $q_j = q_{j+n} = r$.

We can show the following.

Lemma 3.1.10. *There exists a run of* Γ *satisfying* $\neg\varphi_f$ *iff there exists a satisfiable symbolic lasso.*

Proof: For the *only-if* part, suppose there is a run satisfying $\neg\varphi_f$. By Lemma 3.1.1, there exists a satisfiable symbolic run $\{\psi_i\}_{i\geq 0}$ whose corresponding sequence $\{\sigma_i\}_{i\geq 0}$ of truth assignments to P is accepted by $B_{\neg\varphi}$. Thus, there exists an accepting run $\{q_i\}_{i\geq 0}$ of $B_{\neg\varphi}$ on $\{\sigma_i\}_{i\geq 0}$. Let r be an accepting state of $B_{\neg\varphi}$ occurring infinitely often in the run. Since there are finitely many inherited constraints there exists an infinite subset I of integers such that $red(\eta_j^*)$ is the same for all $j \in I$ and $q_j = r$ for all $j \in I$. A simple pigeonhole argument further shows that there is an infinite subset J of I for which:

- for each $u, v \in \bar{x}$, $[u_{j_1}] = [v_{j_1}]$ iff $[u_{j_2}] = [v_{j_2}]$ for all $j_1, j_2 \in J$, and
- for each $u \in \bar{x}$ and $j_1, j_2 \in J$, $[u_{j_1}] = [u_{j_2}]$ or $[u_{j_1}] \neq [v_{j_2}]$ for all $v \in \bar{x}$.

Now pick arbitrary $j, j+n \in J$. Clearly, $\{\psi_i\}_{i<j+n}$ is satisfiable and is a symbolic lasso.

Consider the *if* part. Let $\lambda = \{\psi_i\}_{i<j+n}$ be a satisfiable symbolic lasso. Let D be an instance of $\mathscr{DB}$ for which $Runs_D(\lambda) \neq \emptyset$. We show that

(†) there exists $\{\rho_i\}_{i\leq j+n} \in Runs_D(\lambda)$ such that $\rho_j = \rho_{j+n}$.

Observe that (†) suffices to establish the *if* part of the lemma. Indeed, if (†) holds then the sequence $\{\rho_i\}_{i\leq j}(\{\rho_i\}_{j<i\leq j+n})^\omega$ is an actual run on D of the symbolic run $\{\psi_i\}_{i\leq j}(\{\psi_i\}_{j<i\leq j+n})^\omega$, which satisfies $\neg\varphi_f$.

Consider (†). Let $\bar{y}$ consist of the variables $y \in \bar{x}$ such that $[y_j] = [y_{j+n}]$ and let $\bar{v} = \bar{x} - \bar{y}$. Consider $\eta_j(\bar{y}_j, \bar{v}_j)$. Intuitively, $\bar{y}$ consist of the variables that are preserved from configuration j to $j+n$, while $\bar{v}_j$ and $\bar{v}_{j+n}$ are only related via $\bar{y}$. This together with the fact that $red(\eta_j^*(\bar{x})) = red(\eta_{j+n}^*(\bar{x}))$ will allow us to generate an actual run with the same configurations at j and $j+n$.

We next make this argument more precise. Recall the construction in the proof of Lemma 3.1.3. Consider $[\eta_{j+n}]$, with the augmented set of free variables $[\bar{y}], [\bar{v}_j], [\bar{v}_{j+n}]$.

Let $\bar{u}$ be its quantified variables. Let $G = G_{j+n}$ and consider η^*_{j+n}. Note first that, because of feedback-freedom, $[\bar{v}_j]$ and $[\bar{v}_{j+n}]$ occur in distinct connected components of $G|(\bar{u} \cup [\bar{v}_j] \cup [\bar{v}_{j+n}])$ (indeed, any path connecting a node in $[\bar{v}_j]$ to a node in $[\bar{v}_{j+n}]$ must go through a variable in $[\bar{y}]$). Let H_j consist of the connected components of $G|(\bar{u} \cup [\bar{v}_j] \cup [\bar{v}_{j+n}])$ containing nodes in $[\bar{v}_j]$, and H_{j+n} consist of those containing nodes in $[\bar{v}_{j+n}]$. Let $\chi_j([\bar{y}],[\bar{v}_j])$ and $\chi_{j+n}([\bar{y}],[\bar{v}_{j+n}])$ be the subformulas of $[\eta_{j+n}]$ containing variables in $[\bar{y}]$ and in H_j and H_{j+n}, respectively.

Consider now $[\eta_j]([\bar{y}],[\bar{v}_j])$ and let K_j be the subgraph of G_j consisting of the connected components of $G_j|(\bar{s} \cup [\bar{v}_j])$, that contain nodes in $[\bar{v}_j]$, where $\bar{s}$ are the quantified variables of $[\eta_j]$. Let $\xi_j([\bar{y}],[\bar{v}_j])$ be the subformula of $[\eta_j]$ containing nodes in $[\bar{y}]$ and K_j. As in the proof of Lemma 3.1.3, one can construct formulas $\chi^*_j([\bar{y}],[\bar{v}_j])$, $\chi^*_{j+n}([\bar{y}],[\bar{v}_{j+n}])$ and $\xi^*_j([\bar{y}],[\bar{v}_j])$.
Intuitively, $\xi^*_j([\bar{y}],[\bar{v}_j])$ is the component of the inherited constraint η^*_j constraining $[\bar{v}_j]$ and $[\bar{y}]$. Similarly, $\chi^*_{j+n}([\bar{y}],[\bar{v}_{j+n}])$ plays the same role in the inherited constraint η^*_{j+n}. From the fact that $red(\eta^*_j([\bar{x}])) = red(\eta^*_{j+n}([\bar{x}]))$, it easily follows that
$red(\xi^*_j([\bar{x}])) = red(\chi^*_{j+n}([\bar{x}]))$. In particular, $\xi^*_j([\bar{x}]) \sim \chi^*_{j+n}([\bar{x}])$.

Now consider how the formulas $\xi^*_j([\bar{y}],[\bar{v}_j])$ and $\chi^*_j([\bar{y}],[\bar{v}_j])$ differ. Intuitively, χ^*_j adds to ξ^*_j additional constraints imposed by the segment of λ between configuration j and $j+n$. Specifically, it is easily seen that

$$(\ddagger)\ \chi^*_j([\bar{y}],[\bar{v}_j]) = \xi^*_j([\bar{y}],[\bar{v}_j]) \wedge \mu([\bar{y}],[\bar{v}_j])$$

for some $\mathrm{CQ}^{\neg}$ formula μ.

Consider a run $\rho = \{\rho_i\}_{i \leq j+n} \in Runs_D(\lambda)$. Let μ be the assignment to the variables in λ yielding ρ. We denote by $[\mu]$ the assignment to equivalence classes defined by $[\mu]([y_i]) = \mu(y_i)$ for each $y \in \bar{x}$ and $i \leq j+n$ (this is well defined because μ must satisfy all formulas in λ, including the equalities). We modify μ as follows, yielding a new assignment ν. The assignment ν is the same as μ for all variables v for which $[v]$ is not in H_{j+n}. From $(\ddagger)$ it follows that $D \cup \mathscr{C} \models \xi^*_j([\mu]([\bar{x}_j]))$. Since $\xi^*_j([\bar{x}]) \sim \chi^*_{j+n}([\bar{x}])$, it follows that $D \cup \mathscr{C} \models \chi^*_{j+n}([\mu]([\bar{x}_j]))$. Let ν be defined on the variables in H_{j+n} by setting $\nu(\bar{x}_{j+n}) = \mu(\bar{x}_j)$ and extending $[\mu]([\bar{x}_j])$ to all variables in H_{j+n} by a choice of witnesses to the quantified variables in χ^*_{j+n} satisfying all quantifier-free subformulas of χ^*_{j+n}. It is clear that $D \cup \mathscr{C} \models \psi_i(\nu(\bar{x}_i, \bar{x}_{i+1}))$ for each $i < j+n$. It

follows that $\{\nu(\bar{x}_i)\}_{i\leq j+n}$ is in $Runs_D(\lambda)$. By construction, $\nu(\bar{x}_j) = \nu(\bar{x}_{j+n})$ so the run satisfies (†). □

Decision procedure The above development provides a decision procedure for satisfaction of LTL-FO properties of feedback-free systems, which we outline next. Recall that the LTL-FO property can be assumed to have no global variables by Lemma 2.2.3. The input to the algorithm is an artifact system $\Gamma = \langle \mathscr{A}, \Sigma, \Pi \rangle$ and LTL-FO property φ_f over $\mathscr{A}$ such that (Γ, φ_f) is feedback-free. We begin by constructing the sets of formulas $\Delta = \Delta_\Sigma \cup \Delta_{\varphi_f}$ and the Büchi automaton $B_{\neg\varphi}$. We use a procedure Büchi-Next which, given a state p of $B_{\neg\varphi}$ and a truth assignment σ to the propositions of φ returns one next state of $B_{\neg\varphi}$.

The algorithm non-deterministically searches for a satisfiable symbolic lasso as follows:

1. flag := 0;
2. initialize a symbolic run prefix ρ to $\{\psi_0\}$, with corresponding truth assignment σ_0 to the propositions of φ and set s to some output of Büchi-Next(q_0, σ_0) for some initial state q_0 of $B_{\neg\varphi}$;
3. set γ to $red(\eta^*(\rho))$, where $\eta(\rho)$ is the inherited constraint for the prefix ρ;
4. initialize the set $\mathscr{R}$ of reduced configurations to $\{(s, \gamma)\}$;
5. if flag = 0 and s is an accepting state of $B_{\neg\varphi}$ then non-deterministically continue or set $(\bar{s}, \bar{\gamma}) := (s, \gamma)$, flag:= 1, and $\mathscr{R} := \emptyset$;
6. non-deterministically generate from Δ a symbolic transition ψ with corresponding truth assignment σ to the propositions in φ;
7. set s to Büchi-Next(s, σ), $\gamma := red(\eta^*(\rho\psi))$, and $\rho := \rho.\psi$;
8. if $(s, \gamma) \in \mathscr{R}$ then output NO and stop; otherwise, set $\mathscr{R} := \mathscr{R} \cup \{(s, \gamma)\}$;
9. if flag = 1, $(s, \gamma) = (\bar{s}, \bar{\gamma})$, and $\bar{\gamma}$ is satisfiable, output YES and stop. Otherwise, go to 5.

To see that this provides a decision procedure, we need to show that (i) it terminates and provides the correct answer (i.e. it outputs YES on some execution on (Γ, φ_f) iff $\Gamma \models \varphi_f$), and (ii) the satisfiability test in step 9 is effective.

To see (i), note that, from the definition of inherited constraint $\eta(\rho)$ and the equivalence with $red(\eta^*(\rho))$, it follows that:

(§) if ρ_1, ρ_2 are satisfiable prefixes of symbolic runs such that $red(\eta^*(\rho_1)) = red(\eta^*(\rho_2))$ and ψ is a symbolic transition, then

- $\rho_1.\psi$ is satisfiable iff $\rho_2.\psi$ is satisfiable, and
- the sets of non-deterministically constructed $red(\eta^*(\rho_1.\psi))$ and $red(\eta^*(\rho_2.\psi))$ are equal.

This means that the search for a symbolic lasso can be confined to prefixes with no repeated configurations (s, γ) apart from the knot $(\bar{s}, \bar{\gamma})$, which is enforced by step 8. Since there are finitely many reduced inherited constraints for symbolic runs of (Γ, φ_f) this bounds the running time of the above procedure.

For (ii), we discuss the procedure for checking satisfiability of reduced inherited constraints.

We show that satisfiability of a CQ$^\neg$ formula over $\mathscr{D}\mathscr{B} \cup \mathscr{C}$ can be decided in a modular fashion, by independently checking satisfiability of formulas over $\mathscr{D}\mathscr{B}$ and over $\mathscr{C}$. The significance of the result is that it enables a generic model checking algorithm that takes as parameter the fixed interpretation of $\mathscr{C}$, as long as it comes with a domain-specific satisfiability checker $SAT_{\mathscr{C}}$. $SAT_{\mathscr{C}}$ is called as a black box by the model checker.

More precisely, let q be a prenex normal form CQ$^\neg$ formula over $\mathscr{D}\mathscr{B} \cup \mathscr{C}$: $q = \exists \bar{z} \xi(\bar{u})$ where ξ is quantifier-free, of free variables $\bar{u}$ ($\bar{z} \subseteq \bar{u}$). Let $\xi|_{\mathscr{D}\mathscr{B}}$ and $\xi|_{\mathscr{C}}$ be the restrictions of ξ to schemas $\mathscr{D}\mathscr{B}$ and $\mathscr{C}$, respectively, and denote with $\bar{y} = vars(\xi|_{\mathscr{D}\mathscr{B}}) \cap vars(\xi|_{\mathscr{C}})$ the variables they have in common. Denote with $\bar{c}$ all constants appearing in $\xi|_{\mathscr{D}\mathscr{B}}$. Recall that an equality type $eq(\bar{t})$ over a set $\bar{t}$ of terms (variables or constants) is a satisfiable conjunction of equality and non-equality atoms over $\bar{t}$ such that every pair of terms from $\bar{t}$ occurs in some atom of eq. The following claim follows immediately from the preservation of CQ$^\neg$ formulas under isomorphisms:

(‡) q is satisfiable if and only if there exists an equality type $eq(\bar{y},\bar{c})$, such that $\xi|_{\mathscr{C}} \wedge eq(\bar{y},\bar{c})$ and $\xi|_{\mathscr{D}\mathscr{B}} \wedge eq(\bar{y},\bar{c})$ are satisfiable.

Notice that the satisfiability check for $\xi|_{\mathscr{C}} \wedge eq(\bar{y},\bar{c})$ in the claim is domain-specific (i.e. depends on the fixed interpretation of $\mathscr{C}$), being settled by calling $SAT_{\mathscr{C}}$. Also recall that $\xi|_{\mathscr{D}\mathscr{B}} \wedge eq(\bar{y},\bar{c})$ is an existentially-quantified formula. Therefore its satisfiability reduces to checking that: (a) no pair of terms appears both in an equality and a non-equality atom, and (b) no tuple of terms appears both in a positive and a negative literal with the same relational symbol. This establishes decidability of verification for LTL-FO properties of feedback-free systems, completing the proof of Theorem 3.0.1.

3.1.2 Complexity

The complexity analysis of the decision procedure involves several orthogonal components:

Computing reduced inherited constraints This involves the recursive construction in the proof of Lemma 3.1.3. It is easily seen that it requires polynomial time in the size of inherited constraint γ, which is bounded by the size of the the symbolic run prefix ρ.

Performing the satisfiability check (in step 9 of the decision procedure), which consists in

(i) Retrieving the prenex normal form of $\bar{\gamma}$, which is $\eta(\rho)$, then guessing an equality type eq on the number of variables $\eta(\rho)|_{\mathscr{C}}$ and $\eta(\rho)|_{\mathscr{D}\mathscr{B}}$ have in common, and the number of constants mentioned in $\eta(\rho)|_{\mathscr{D}\mathscr{B}}$. This can be done in NP in the number of common variables and of constants, which is bounded by the size of $\eta(\rho)$.

(ii) Running $SAT_{\mathscr{C}}$ on $\eta(\rho)|_{\mathscr{C}} \wedge eq$. This step depends on the fixed interpretation of $\mathscr{C}$. If $\mathscr{C}$ is interpreted as linear arithmetic inequalities, the test reduces to solving a linear programming problem. This yields polynomial time in the size of $\eta(\rho)|_{\mathscr{C}} \wedge eq$ [Kar84], which in turn is polynomially (quadratically) bounded by the size of $\eta(\rho)$. Indeed, each pair of terms in $\eta(\rho)|_{\mathscr{C}}$ must be related explicitly in eq by either an equality or a non-equality atom.

(iii) Checking satisfiability of $\eta(\rho)|_{\mathscr{D}\mathscr{B}} \wedge eq$. This is doable in polynomial time in the size of $\eta(\rho)|_{\mathscr{D}\mathscr{B}} \wedge eq$, which is again polynomially bounded by the size of $\eta(\rho)$.

The search for the symbolic lasso This step is polynomial in the number of reduced inherited constraints and the states of $B_{\neg\varphi}$ visited during the search. The test that the current inherited constraint γ is the same as the knot candidate $\bar{\gamma}$ is polynomial in the size of $\bar{\gamma}$.

Since the size of reduced inherited constraints γ is upper bounded by the length of the run ρ, which in turn is bounded by the number of distinct reduced inherited constraints, by Lemma 3.1.3 we obtain:

Proposition 3.1.11. *Static verification for feedback-free pairs of LTL-FO properties and artifact systems is decidable in time upper bounded by $h(k^2)$, for some $h \in Hyp$.*

3.1.3 Subclasses with Improved Upper Bounds

The above analysis shows that the complexity of the decision procedure is dominated by the number of distinct inherited constraints, which upper bounds the symbolic run length. We identify next three sub-classes of feedback-free artifact systems that occur naturally and lead to a better bound.

Bounded width The construction in the proof of Lemma 3.1.3 introduces the useful notion of *width of a set of variables* in a symbolic run. Recall that the width $w(\bar{y})$ of $\bar{y}$ is defined as $max\{|\bar{v}| \mid \bar{v} \subseteq \bar{x}_i$, and each $v \in \bar{v}$ is in an equivalence class $y \in \bar{y}\}$. By extension, the width of a subgraph of G_j is the width of its set of nodes.

Definition 3.1.12. *We say that (Γ, φ_f) has width bounded by w if it is feedback-free, and if for each symbolic run, the width of each connected component of G_j is bounded by w for each $j \geq 0$.*

Intuitively, the width bound indicates the maximum number of variables in $\bar{x}$ that are mutually related in a configuration of a symbolic run. We can show the following.

Corollary 3.1.13. *If (Γ, φ_f) has width bounded by w, then the number of distinct reduced inherited constraints is bounded by $(h(w^2))^k$, where $h \in Hyp$.*

Proof: The bound is an immediate consequence of the construction in the proof of Lemma 3.1.3, and the fact that constraints corresponding to distinct connected components of G_j are independent. □

Thus, Corollary 3.1.13 provides a smaller bound on the number of reduced inherited constraints if the connected components generated in symbolic runs have small width.

Linear propagation An artifact system and a property exhibit *linear propagation* if the feedback-freedom restriction is satisfied for a more restrictive definition of variable equivalence classes. Equivalence classes are generated exclusively by equalities of the form $x = x'$, and any other equalities are treated as arithmetic constraints.

Note that every linear-propagation equivalence class involves the values of a *single* variable. In the graphical representation of symbolic transition templates and runs, all equality edges are horizontal. This is the case in our running example.

Proposition 3.1.14. *Let (Γ, φ_f) exhibit linear propagation. The number of distinct reduced inherited constraints in configurations of symbolic runs is bounded by $h(k)$, for some function $h \in Hyp$.*

Proof: We claim that the quantifier rank of inherited consrtaints is bounded by k, which immediately implies the upper bound in the proposition's statement.

The proof is similar to the proof of Lemma 3.1.3, except the induction shows that β can be rewritten with quantifier rank bounded by $fp(\bar{y})$, with fp defined as follows.

In the notation of the proof of Lemma 3.1.3, define the *footprint* of $\bar{y}$ in $G_\beta|\bar{y}$ as the set of artifact variables whose values appear in the equivalence classes $\bar{y}$: $fp(\bar{y}) = \{x | x \in \bar{x}, y \in \bar{y}, [x_i] \in y \text{ for some } i\}$. Clearly, $|fp(\bar{y})| \leq k$.

We prove by induction on the size of $fp(\bar{y})$ that the formula corresponding to $\exists \bar{y} \beta(\bar{y}, \bar{s})$ can be rewritten with quantifier rank bounded by $|fp(\bar{y})|$. As in the proof of Lemma 3.1.3, consider $\bar{v} = \bar{y} - \bar{y}_{max}$, and let $H_1, \ldots, H_c$ be the connected components of $G_\beta|\bar{v}$ and $\bar{y}_r$ the set of nodes on H_r, for all $1 \leq r \leq c$.

Now observe that linear propagation ensures that each equivalence class consists of successive values of the same variable, $[y] = \{y_l | l \in span([y])\}$. Also observe that, if $[x_i] \in \bar{y}_{max}$, then for all l and all $1 \leq r \leq c$, $[x_l] \notin \bar{y}_r$ (otherwise the span maximality

of $\bar{y}_{max}$ is contradicted). It follows that $fp(\bar{y}_r) \subseteq fp(\bar{y}) - fp(\bar{y}_{max})$, so $|fp(\bar{y}_r)| \leq |fp(\bar{y})| - |fp(\bar{y}_{max})|$, and the induction hypothesis applies to each H_r. □

Corollary 3.1.15. *If linear-propagation pair (Γ, φ_f) has width bounded by w, then there are at most $(h(w))^k$ distinct reduced inherited constraints, for some $h \in Hyp$.*

Proof: The proof follows immediately from Proposition 3.1.14 and the observation that constraints corresponding to distinct connected components are independent. □

Acyclicity Recall that feedback-freedom restricts the way in which the value of variable x at step j can be connected to the value of x at preceding step i. We investigate a more stringent restriction, which disallows any such connection (except for preservation of the value of x from i to j).

Definition 3.1.16. (Γ, φ_f) *is* acyclic *iff for every symbolic run prefix $\rho = \{\psi_i\}_{i \leq n}$, if x_i and x_j are connected in G_ρ, then $[x_i] = [x_j]$.*

Note that acyclicity trivially implies feedback-freedom: in Definition 2.3.2, the role of y is played by x_i.

Proposition 3.1.17. *Let (Γ, φ_f) be acyclic. The number of distinct reduced inherited constraints in configurations of the same symbolic run is bounded by a doubly-exponential function of k.*

Proof: We recall from the proof of Lemma 3.1.3 the notation G_j for the graph on equivalence classes corresponding to symbolic run prefix $\rho|_j$. We also extend the notion of span beyond equivalence classes to arbitrary subgraphs of G_j, in the natural way.

We claim that any connected component H of G_j has at most k distinct equivalence classes.

Indeed, given an equivalence class $[e]$ in H, denote with

$$vars([e]) = \{x | x \in \bar{x}, l \in span([e]), x_l \in [e]\},$$

i.e. the set of artifact variables whose value belongs to $[e]$ at some step. Observe that acyclicity implies that there is no $i, j \in span(H)$ with $[x_i] \neq [x_j]$ in H. Therefore, the equivalence classes in H have pairwise disjoint *vars* sets. This implies the claim.

By the claim, the corresponding sub-formula of the inherited constraint can be written using at most k variables. These are free if the span of their equivalence class includes j, and existentially quantified otherwise.

The number of distinct inherited constraints corresponding to connected components of G_j can be upper bounded as follows. By the claim, the number of distinct literals over k variables is upper bounded by $(2|\mathscr{DB} \cup \mathscr{C}|)^k$ (where $|\mathscr{DB} \cup \mathscr{C}|$ denotes the sum of the arities of the relation in $\mathscr{DB} \cup \mathscr{C}$). The number of distinct conjunctions thereof is bounded by $2^{2|\mathscr{DB} \cup \mathscr{C}|^k}$. Since the number of distinct choices for the subset of existentially quantified variables is bounded by 2^k, we obtain a bound of $N = 2^{k(2|\mathscr{DB} \cup \mathscr{C}|^k)}$ on the number of distinct inherited constraint sub-formulas that correspond to connected components.

The proposition now follows from the above upper bound and the observation that the independence of connected components allows re-using variables across them. □

To illustrate the difference between acyclicity and feedback freedom, consider again our running example. As discussed earlier, feedback freedom allows changing the shipment type unboundedly many times for the same product. In contrast, acyclicity disallows such runs. If the customer wants to change the shipment type, she must forget all her choices and starts filling the order from scratch (select a product again, then a shipment type). This puts the two shipment type choices in disconnected components of the computation graph.

3.2 Introducing Dependencies

We show next that model checking for feedback-free pairs of LTL-FO properties and artifact systems is decidable even in the presence of expressive database integrity constraints modeled by dependencies.

Given a set $\mathscr{I}$ of dependencies on the database schema $\mathscr{DB}$, we say that an artifact system Γ satisfies an LTL-FO sentence φ *under* $\mathscr{I}$, denoted $\Gamma \models_{\mathscr{I}} \varphi$, if for every database D satisfying $\mathscr{I}$ and every run ρ of Γ on D, φ holds on ρ.

We next establish decidability under a set of dependencies, provided that the

chase with these dependencies terminates. The chase is a fundamental algorithm that has been widely used in databases. It takes as input an initial instance A and a set of dependencies $\mathscr{I}$ and produces (if it terminates, which is not guaranteed), a finite model of $\mathscr{I}$ and A that satisfies a universality property (see [AHV95] for details).

3.2.1 Relevant Chase Properties

The chase [AHV95] is a fundamental algorithm that has been widely used in databases. It takes as input an initial instance A and a set of dependencies $\mathscr{I}$ and, if it terminates (which is not guaranteed), its result is a finite instance U satisfying:

(a) U is a model of $\mathscr{I}$ and A

(we say that U is a model of $\mathscr{I}$ and A if U satisfies the dependencies $\mathscr{I}$ and there is a homomorphism from A to U), and

(b) U is *universal* for $\mathscr{I}$ and A: that is, it has a homomorphism into every model of $\mathscr{I}$ and A.

We call a finite instance with these properties a *universal model* for $\mathscr{I}$ and A.

At every *chase step*, the algorithm identifies a violation of some dependency in $\mathscr{I}$ and modifies A to remove this violation (the modification is minimal in a certain sense, and it possibly introduces new violations). When several violations exist, the choice of the one to remove is non-deterministic, and the sequence of such choices induces a sequence of chase steps, called a *chase sequence*. The sequence is *terminating* if it reaches an instance that satisfies $\mathscr{I}$.

We recall from [DNR08] an extension of the chase to *disjunctive embedded dependencies (DEDs)*. Here, we are only interested in the particular case when the DED conclusion consists of the empty (always false) disjunction $\bot$. As soon as a chase step derives $\bot$, the chase sequence terminates, yielding the result $\bot$. We then say that the chase *fails*. A terminating chase sequence is a finite sequence of chase steps which either fails or yields an instance that satisfies all dependencies.

3.2.2 Verification With Dependencies

We borrow from [MSWL10] the notation $CT_{\forall\exists}$ for the class of dependency sets $\mathscr{I}$ such that for every instance A there exists a terminating chase sequence of A with $\mathscr{I}$.

Theorem 3.2.1. *It is decidable, given artifact system Γ and an LTL-FO sentence $\forall\bar{y}\varphi_f$ such that $(\Gamma, \forall\bar{y}\varphi_f)$ is feedback-free, and given set $\mathscr{I} \in CT_{\forall\exists}$ of dependencies on $\mathscr{DB}$, whether Γ satisfies $\forall\bar{y}\varphi_f$ under $\mathscr{I}$.*

While membership of a set of dependencies in $CT_{\forall\exists}$ is in general known to be undecidable (see for instance [DNR08]), recent research has proposed sufficient syntactic restrictions. Examples include *weak acyclicity* [FKMP03], stratification [DNR08], and the *termination hierarchy* [MSWL10] which is a hierarchy of successive relaxations of weak acyclicity and stratification that nevertheless suffice for the existence of a terminating chase sequence.

Corollary 3.2.2. *If $\mathscr{I}$ lies in the terminaton hierarchy and $(\Gamma, \forall\bar{y}\varphi_f)$ is feedback-free, then $\Gamma \models_{\mathscr{I}} \forall\bar{y}\varphi_f$ is decidable.*

The proof of Theorem 3.2.1 is given after introducing a few useful definitions and results.

Let $q(\bar{u})$ be a CQ$^\neg$ formula over $\mathscr{DB} \cup \mathscr{C}$ with free variables $\bar{u}$. We say that q is *$\mathscr{I}$-satisfiable* if there exists $D \models \mathscr{I}$ and a valuation ν of $\bar{u}$ such that $D \cup \mathscr{C} \models q(\nu)$. We say that symbolic run ρ is *$\mathscr{I}$-satisfiable* if there exists $D \models \mathscr{I}$ such that $Runs_D(\rho) \neq \emptyset$. The definition extends naturally to prefixes of symbolic runs, and in particular to symbolic lassos.

Decision procedure The decision procedure we exhibit in proving Theorem 3.2.1 is the one presented in Section 3.1 for the dependency-free case, modified as follows: in step 9, the test of satisfiability of $\bar{\gamma}$ is replaced with an $\mathscr{I}$-satisfiability test.

The remainder of the section outlines the proof that the above modification yields a decision procedure for model checking under dependencies, provided that the set of dependencies lies in the termination hierarchy.

We extend Claim (‡) from Section 3.1 to the presence of dependencies. We first show that $\mathscr{I}$-satisfiability of a CQ$^\neg$ formula over $\mathscr{D}\mathscr{B}\cup\mathscr{C}$ can be decided in a modular fashion, by independently checking satisfiability of formulas over $\mathscr{D}\mathscr{B}$ and over $\mathscr{C}$.

More precisely, let q be a prenex normal form CQ$^\neg$ formula over $\mathscr{D}\mathscr{B}\cup\mathscr{C}$: $q = \exists\bar{z}\xi(\bar{u})$ where ξ is quantifier-free, of free variables $\bar{u}$ ($\bar{z}\subseteq\bar{u}$). Let $\xi|_{\mathscr{D}\mathscr{B}}$ and $\xi|_{\mathscr{C}}$ be the restrictions of ξ to schemas $\mathscr{D}\mathscr{B}$ and $\mathscr{C}$, respectively, and denote with $\bar{y} = vars(\xi|_{\mathscr{D}\mathscr{B}})\cap vars(\xi|_{\mathscr{C}})$ the variables they have in common. Denote with $\bar{c}$ all constants appearing in $\xi|_{\mathscr{D}\mathscr{B}}$ or $\mathscr{I}$.

Lemma 3.2.3. *q is $\mathscr{I}$-satisfiable if and only if there exists an equality type $eq(\bar{y},\bar{c})$, such that $\xi|_{\mathscr{C}}\wedge eq(\bar{y},\bar{c})$ is satisfiable and $\xi|_{\mathscr{D}\mathscr{B}}\wedge eq(\bar{y},\bar{c})$ is $\mathscr{I}$-satisfiable.*

Proof: Let's denote the fact that there is a chase sequence s from q to q' with $q \overset{s}{\Rightarrow} q'$.

(i) If $\mathscr{I}\in CT_{\forall\exists}$, then there is a terminating chase sequence s_1 of q. By definition, s_1 is finite and if $q\overset{s_1}{\Rightarrow}q'$, q' satisfies all dependencies in $\mathscr{I}$ (as $\mathscr{I}$ does not mention $\bot$).

Observe that the chase steps with dependencies from $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ only introduce $\bot$, which is not mentioned in $\mathscr{I}$. Therefore, chase steps with dependencies from $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ cannot enable chase steps with dependencies from $\mathscr{I}$. Also observe that the chase with $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ must terminate after at most one step (either no dependency in $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ applies, or if one applies, then the chase fails).

Let s_2 be a chase sequence with $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ such that $q'\overset{s_2}{\Rightarrow}q''$. By the above observation, s_2 is terminating, and has length at most 1. If the length is 0, then $q''=q'$, so q'' satisfies $\mathscr{I}\cup\mathscr{I}_{\mathscr{D}\mathscr{B}}$. If the length is 1, then $q''=\bot$, and the chase fails. Therefore s_1,s_2 is a terminating chase sequence with $\mathscr{I}\cup\mathscr{I}_{\mathscr{D}\mathscr{B}}$, thus witnessing that $\mathscr{I}\cup\mathscr{I}_{\mathscr{D}\mathscr{B}}\in CT_{\forall\exists}$.

(ii) By (i), there is a terminating chase sequence s of q with $\mathscr{I}\cup\Delta_D B$. Denote with $s|_i$ the prefix of length i of s, with s_i the i'th chase step, and with q_i the result of the first i chase steps: $q\overset{s|_i}{\Rightarrow}q_i$.

If s does not fail, then the result of s, when viewed as an instance, witnesses satisfiability.

Suppose that the chase fails. Then we prove by contradiction that there is no model that satisfies q and $\mathscr{I}$.

For assume that such a model D exists. Let n be the length of s. Since the chase fails, and failure must occur after the first application of any dependency in $\mathscr{I}_{\mathscr{D}\mathscr{B}}$, $s|_{n-1}$

uses only dependencies from $\mathscr{I}$, and s_n must use a dependency $\delta \in \mathscr{I}_{\mathscr{D}\mathscr{B}}$. The premise of δ must map homomorphically into q_{n-1}. Call this homomorphism h.

Moreover, let $q = q^- \wedge q^+$, where q^- is the subquery of q consisting of all negative literals, and q^+ the subquery consisting of all positive literals.

Since $\mathscr{I}$ mentions only positive literals, the first $n-1$ chase steps apply only to q^+, so $q^-_{n-1} = q^-$ and $q^+ \overset{s|_{n-1}}{\Rightarrow} q^+_{n-1}$. Recall that the premise of all dependencies in $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ consists of a positive literal L and its negation $\neg L$. In particular it follows that h maps L into q^+_{n-1}, and $\neg L$ into q^-.

Notice that the chase of q^+ is standard, involving a positive conjunctive query and positive dependencies. It is known (see [AHV95] for instance) that the standard chase preserves universality. In particular, this means that for every $1 \leq i \leq n-1$, q^+_i has a homomorphic mapping into every instance that satisfies q^+ and $\mathscr{I}$. In particular, there is a homomorpism m from q^+_{n-1} into D.

But then D must contain an image of L under $h \circ m$, which leads to a contradiction: since D satisfies all of q, it must satisfy $\neg h \circ m(L)$ and $h \circ m(L)$ simultaneously. □

As observed in Section 3.1, the satisfiability check for $\xi|_{\mathscr{C}} \wedge eq(\bar{y}, \bar{c})$ in Lemma 3.2.3 is domain-specific (i.e. depends on the fixed interpretation of $\mathscr{C}$), being settled by calling $SAT_{\mathscr{C}}$.

We next show that $\mathscr{I}$-satisfiability of formulas over $\mathscr{D}\mathscr{B}$ reduces to chasing with $\mathscr{I}$ and an appropriately selected set $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ of dependencies whose construction is determined by the schema $\mathscr{D}\mathscr{B}$.

We recall from [DNR08] an extension of the chase to *disjunctive embedded dependencies (DEDs)*. Here, we are only interested in the particular case when the DED conclusion consists of the empty (always false) disjunction $\bot$. As soon as a chase step derives $\bot$, the chase sequence terminates, yielding the result $\bot$. We then say that the chase *fails*. A terminating chase sequence is a finite sequence of chase steps which either fails or yields an instance that satisfies all dependencies.

Define the set of dependencies

$$\mathscr{I}_{\mathscr{D}\mathscr{B}} := \{\delta_{\neq}\} \cup \{\delta^P_{\neg} \mid P \in \mathscr{D}\mathscr{B}\}$$

where

$$\delta_{\neq} \;:\; \forall x \forall y\, x = y \wedge x \neq y \rightarrow \bot$$
$$\delta^{P}_{\neg} \;:\; \forall \bar{x}\, P(\bar{x}) \wedge \neg P(\bar{x}) \rightarrow \bot.$$

Lemma 3.2.4. *If $\mathscr{I} \in CT_{\forall\exists}$, then*

(i) $\mathscr{I} \cup \mathscr{I}_{\mathscr{D}\mathscr{B}} \in CT_{\forall\exists}$, and

(ii) a formula $q \in CQ^{\neg}$ over $\mathscr{D}\mathscr{B}$ is $\mathscr{I}$-satisfiable if and only if the chase of q with $\mathscr{I} \cup \mathscr{I}_{\mathscr{D}\mathscr{B}}$ does not fail.

Remark 3.2.5. *We could have applied the more general decision procedure from [DNR08], for satisfiability of $CQ^{\neg}$ under a set of dependencies with negated literals. The result in [DNR08] is based on chasing with more expressive, disjunctive dependencies, yielding a tree of chase sequences. When applied to our setting, this would result in an exponential number of chase sequences. Lemma 3.2.4 shows that this exponential blow-up can be avoided by resorting to the standard (non-disjunctive) chase, and by exploiting the fact that the dependencies in $\mathscr{I}$ contain only positive literals.*

The following result establishes the decidability of $\mathscr{I}$-satisfiability for finite prefixes of symbolic runs.

Lemma 3.2.6. *It is decidable, given a symbolic run prefix ρ and $\mathscr{I} \in CT_{\forall\exists}$ on $\mathscr{D}\mathscr{B}$, whether ρ is $\mathscr{I}$-satisfiable.*

Proof: Lemmas 3.1.2, 3.2.3 and 3.2.4 imply that the following is a decision procedure for $\mathscr{I}$-satisfiability of ρ. Let η_n be the inherited constraint for configuration n of ρ (in prenex normal form), and ξ be its quantifier-free body. Let $\xi|_{\mathscr{D}\mathscr{B}}$, $\xi|_{\mathscr{C}}$, $\bar{y}$, $\bar{c}$, eq be as in Lemma 3.2.3, and $\mathscr{I}_{\mathscr{D}\mathscr{B}}$ as in Lemma 3.2.4. Return YES if and only if $SAT_{\mathscr{C}}$ returns YES on $\xi|_{\mathscr{C}} \wedge eq$ and the chase of $\xi|_{\mathscr{D}\mathscr{B}} \wedge eq$ with $\mathscr{I} \cup \mathscr{I}_{\mathscr{D}\mathscr{B}}$ does not fail. □

Complexity The complexity upper bound obained in the absence of dependencies is virtually unaffected by the presence of sets of dependencies from the termination hierarchy.

First, recall that the satisfiability check for the restiction of the inherited constraints to $\mathscr{C}$ is settled by calling $SAT_{\mathscr{C}}$, thus inheriting the complexity of the particular instantiation of $\mathscr{C}$.

Second, the complexity of the $\mathscr{I}$-satisfiability check for the inherited constraints restricted to $\mathscr{DB}$ inherits the complexity of the chase with sets of dependencies from the termination hierarchy.

This complexity is polynomial in the size of the inherited constraint, with the polynomial's degree bounded by the size of $\mathscr{I}$ [MSWL10, DNR08, FKMP03]. The bound is very conservative, and a refined analysis shows that it depends on the longest path in a graph that reflects how a chase step with one dependency can trigger another.

Given that we expect multiple verification instances over the same database schema with integrity constraints, a reasonable assumption (often adopted in the literature) is to consider schema and dependencies fixed. This yields a truly polynomial complexity of the chase. The same truly polynomial complexity holds, even if $\mathscr{DB}$ and $\mathscr{I}$ are not fixed, if the dependencies in $\mathscr{I}$ are *equality-generating dependencies* [AHV95]. It also holds if only $\mathscr{DB}$ is fixed but not $\mathscr{I}$, if it consists only of *full* dependencies [AHV95]. Equality-generating dependencies allow only equality atoms in the conclusion, and capture as particular cases the class of functional dependencies. Full dependencies contain no existentially quantified variables.

Acknowledgement

Alin Deutsch and Victor Vianu co-authored this chapter.

4 Improved complexity upper bounds with key dependencies

The verification technique presented in Chapter 3 extends the allowed expressivity of business process specifications to the use of generic data dependencies and arithmetic constraints. It does so, however, at the cost of a very worst-case complexity. Moreover, and even more importantly, the technique retains its high worst-case complexity even when no data dependencies or arithmetic is used (a case in which we are aware of a PSPACE upper bound [DHPV09]).

In the present chapter we restrict the analysis to key dependencies and present an alternative verification technique that, while based on the theory developed in Chapter 3, is an improvement for a number of aspects:

- it provides an EXPSPACE upper bound in the case of verification on feedback-free artifact systems that contain no arithmetic and unary keys;
- it has a PSPACE upper bound in case of constant navigational complexity (an interesting and common class of business processes);
- it allows us to relax the feedback-freedom definition;
- it allows graceful scaling of worst-case complexity, i.e. EXPSPACE for artifact systems with unary keys, and, for artifact systems with arithmetic and generic constraints, the upper bound is hyperexponential only in the cardinality of the variables involved in arithmetic constraints.

In order to ease the presentation, we start introducing the main ideas by presenting a technique that performs verification on specifications with no keys and no arith-

metic in PSPACE. This technique does not introduce any new results, as [DHPV09] already provides a way to check those properties. However, it introduces a way to perform verification that will be the corner stone of our more general technique incorporating single-attribute keys.

In this chapter we will make use of the following naming conventions. We call $\bar{x}_i$ the variables in instant i. We denote as η_j^k the formula $\exists \bar{z}(\eta(\langle\psi_i\rangle_{i\leq k}))$, where $\bar{z}$ is the set of all variables in $\eta(\langle\psi_i\rangle)$ except $\bar{x}_j$ and $\bar{x}_k$. We then call $[\eta_j^k]$ the formula that uses the equivalence classes of the variables in η_j^k as variables (and thus have no equivalence atoms). $[\eta_j^k]$ is called *cycle inherited constraints*. Also, when clear from the context, we use the term 'inherited constraints' both for standard and cycle inherited constraints, and refer as 'free variables' to all variables that are not existentially quantified. In this context we refer to $\bar{x}_i$ as the *equivalence classes* of the variables in instant i. Also, we assume for ease of presentation that symbolic transitions do not contain constants. The extension for constants is easy, and does not change the computational complexity of the technique presented. We will discuss the introduction of constants at the end of the chapter.

4.1 Alternative technique for artifact systems with no dependencies and no arithmetic

The technique is based on the following lemma.

Lemma 4.1.1. *In an artifact system and property with only key constraints, there exists a run ρ satisfying $\neg\varphi$ iff*

1. *there exists a symbolic run prefix $\langle\psi_i\rangle_{i<j+n}$ s.t. $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ is satisfiable, and*

2. *for such $\langle\psi_i\rangle_{i<j+n}$, there exists a run $\{q_i\}_{i\leq j+n}$ of $B_{\neg\varphi}$ on $\{\sigma_i\}_{i\leq j+n}$ (i.e. the run of truth assignments for FO components in $\neg\varphi$ given the run $\{\psi_i\}_{i<j+n}$) such that for some accepting state r. $q_j = q_{j+n} = r$.*

Proof: Clearly, from the definition of inherited constraints, if we have a symbolic run $\langle\psi_i\rangle_{i<j+n}$ as above, there exists a run ρ satisfying $\neg\varphi$.

Now we assume we have a run ρ satisfying $\neg\varphi$. Note that, formally, let D be a database instance for the artifact system, we have that $\rho, D \models \neg\varphi$. Let us consider the infinite set of instants I in which ρ_i is in an accepting state r of $B_{\neg\varphi}$. Now ley us call e an equality type between the attribute values in ρ_i. By pigeonhole principle there is an infinite set of instants with the same equality type e. We call this set I^e. Now we call D^+ the set $D \cup \{\omega\}$. The we define the assignments $\nu_i : \bar{x} \to D^+$ in the following way: $\nu_i(x) = \rho_i(x)$, if $\rho_i(x) \in D$, and $\nu_i(x) = \omega$, otherwise. Since D is finite, it follows that there are a finite number of possible ν_i. By pigeonhole principle there are an infinite number of instants that share the same ν. We call $I^{e,\nu}$ the set of these instant. Let us pick any ν.

Since all instants of $I^{e,\nu}$ share a single ν, any attribute $x \in \bar{x}$ s.t. $\nu(x) = \omega$ is assigned to values not in D in all instants of $I^{e,\nu}$. Let $V(x) = \{\rho_i(x) | i \in I^{e,\nu}\}$ be the set of values assigned to an attribute $x \in \bar{x}$. For all the $x \in \bar{x}$ s.t. $V(x)$ is finite, it is easy to see that (by pigeonhole principle) we can restrinct the set of instants to the ones where each attribute is assigned to the same value. Let $\bar{y}$ be the set of these attributes, and let μ be an assignment from $\bar{y}$ to $\bigcup_{y \in \bar{y}} V(y)$, we call $I^{e,\nu,\mu}$ the set of these instant.

Let us consider the attributes $\bar{z}$ whose $V(x)$ is infinite. Since the values in $V(x)$, $x \in \bar{z}$ are not in D, it follows that the only way the artifact system can force them to be distinct is using $\neq$-inequalities. Also, in each instant every value can be forced to be different from at most $2|\bar{x}| - 1$ possible values (using the primed and unprimed variables in a postcondition). Since this set is finite, it follows that in $[\eta]$ the variables associated in the instant of $I^{e,\nu,\mu}$ to such attributes can be forced different to at most a finite number of other values. Moreover, $\neq$-inequalities do not have any transitive property, it follows that, since attributes in $\bar{z}$ are associated to an infinite set of values we can find instants in $I^{e,\nu,\mu}$ that are far enough such that $[\eta]$ does not force them to be distinct. Let j and $k > j$ be such two instants. It follows that we can build a run ρ' from ρ by substituting all values associated to the variables in the equivalence classes of $\bar{x}_k$ with the corresponding values in ρ_j. More formally, for each $x \in barx$, for each $x \in [x]_k$, $\rho'(x) = \rho(x_j)$. It is easy to see that, by construction, $\rho' \models [\eta]|_j^k(\langle\psi_i\rangle_{i \leq k}) \wedge \bar{x}_j = \bar{x}_k$ and since $j, k \in I$, they also satisfy condition 2. $\square$

The main idea is to explore all possible symbolic runs, while checking the above conditions. Note that, in order to perform this search, it will not be necessary to compute the whole inherited constraints for the symbolic run prefixes.

4.1.1 Reduced form of inherited constraints

The technique in Chapter 3 has an hyper-exponential upper bound because of the size of inherited constraints in feedback free systems. The information in the inherited constraints is necessary to check the existence of runs when we allow in business process specification or in the properties either arithmetic constraints or a set of data dependencies whose chase terminates. By limiting the expressivity, we are able to discard information from the inherited constrains in order to reduce their number. The main idea is that we only keep the information that we need to check satisfiability of $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$. Since we can check satisfiability by looking for contradiction proofs, we define the following concept.

Definition 4.1.2. *Let* $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ *be in prenex form. We call a* no-dep contradiction proof *(or simply* proof*) w.r.t.* $j, j+n$*, a set of the atoms in* $[\eta]$ *of the following kind:*

1. *a pair* $p(\bar{y}) \wedge \neg p(\bar{z})$ *s.t.* $\bar{y}|_{\bar{x}_j \leftarrow \bar{x}_{j+n}} = \bar{z}|_{\bar{x}_j \leftarrow \bar{x}_{j+n}}$*; or*

2. *an atom* $y \neq z$ *s.t.* $y|_{\bar{x}_j \leftarrow \bar{x}_{j+n}} = z|_{\bar{x}_j \leftarrow \bar{x}_{j+n}}$.

Since, by definition, $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})$ is in $CQ^{\neg}$ form, it does not contain any equality atoms, and we assumed we have no constants, the only possible derivation of a contradiction have the form described in Definition 4.1.2. It follows easily that,

Lemma 4.1.3. $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ *is satisfiable iff no subset of its atoms is a contradiction proof.*

This means that the only information we need of the inherited constraints of a symbolic run is the existence of contradiction proofs. In order to define reduced form of inherited constraints we will make use of the concept of renamed formula.

Definition 4.1.4. *Let $\exists\bar{z}(f(\bar{x},\bar{z}))$ be a conjunction of atoms of $[\eta]$ in prenex form, we call the* renamed formula *$ren_{[\eta]}(\exists\bar{z}f(\bar{x},\bar{z})) = \exists\bar{z'}f(\bar{x},\bar{z'})$ a formula that substitutes each existential variable in $\bar{z}$ in $f(\bar{x},\bar{z})$, with a new existential variable in $\bar{z'}$ that does not appear in $[\eta]$.*

We can now define the reduced form:

Definition 4.1.5. *Let $\langle\rho_i\rangle_{i\leq j+n}$ be a symbolic run prefix, we define the* no-dep-proof-reduced *inherited constraints $red^n([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ in the following way. $red^n([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ contains:*

1. *all terms $t(\bar{y})$ s.t. $\bar{y}\subseteq([\bar{x}]_j\cup[\bar{x}]_i)$; and,*
2. *for each proof f of $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})$, $ren_{red^n([\eta])}(f)$ if there is no variable bijection μ s.t. $\mu(x)=x$ for all $x\in[\bar{x_j}]\cup[\bar{x_{j+n}}]$ and $red^n([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ contains $\mu(ren_{red^n([\eta])}(f))$.*

The intuition of the above definition is that we want to keep a single copy for each proof (up to renaming of existential variables) while at the same time discarding the information about the joins between the proofs. The terms in 1) are kept in order to easily compute red^m without fully materializing the inherited constraints (see Lemma 4.1.8).

Now we outline the technique to perform verification. The idea is to create a finite state automaton that recognizes the language of all the symbolic prefixes s.t. $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})\wedge\bar{x}_j=\bar{x}_{j+n}$ is satisfiable and they satisfy $\neg\varphi$. We will then use $red^n([\eta])$ instead of $[\eta]$. For this idea to work we need first to prove that red^n carries enough information to check satisfiability of the whole $[\eta]$.

Lemma 4.1.6. *$[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})\wedge\bar{x}_j=\bar{x}_{j+n}$ is satisfiable iff $red^n([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))\wedge\bar{x}_j=\bar{x}_{j+n}$ is satisfiable.*

Sketch: The only possible form of a contradiction proof is $p(\bar{y})\wedge\neg p(\bar{z})$ with $\bar{y}|_{\bar{x}_j\leftarrow\bar{x}_{j+n}}=\bar{z}|_{\bar{x}_j\leftarrow\bar{x}_{j+n}}$ or $y\neq z$ with $y|_{\bar{x}_j\leftarrow\bar{x}_{j+n}}=z|_{\bar{x}_j\leftarrow\bar{x}_{j+n}}$. It follows by construction that if it appears in $[\eta]$ there is a proof in red^f containing it, and vice versa. □

Note that red^n is polynomially bounded w.r.t. the number of attributes in the artifact system, considering the maximum arity of the database schema fixed.

4.1.2 PSPACE verification

In order to design an efficient procedure for verification, we need a way to compute red^n without computing the whole inherited constraints first. We describe now a way to compute $red^n([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ through a scan of a symbolic prefix $\langle\psi_i\rangle_{i\leq j+n}$, without materializing the full inherited constraints. While scanning the symbolic prefix, we keep only the following formula:

Definition 4.1.7. *We call $active^n([\eta])$, the formula computed while scannning $\langle\psi_i\rangle_{i\leq j+n}$ in the following way:*

- $active^n([\eta](\langle\psi_0\rangle)) = [\eta](\langle\psi_0\rangle)$*;*
- *we keep track of the equivalence classes using the equality atoms in the symbolic transitions;*
- *for all $i <= j$, we build $active^n([\eta]|_i)$ from $active^n([\eta]|_{i-1}) \wedge \psi_i$ and we keep only:*
  1. *the terms $t(\bar{y})$ s.t. $\bar{y} \subseteq [\bar{x}]_i$; and*
  2. *all distinct proofs w.r.t. i,i up to renaming of existential variables (i.e. variables not in $[\bar{x_i}]$).*
- *for all $j < i <= j+n$, we build $active^n([\eta]|_j^i)$ from $active^n([\eta]|_j^{i-1}) \wedge \psi_i$ and we keep only:*
  1. *all terms $t(\bar{y})$ s.t. $\bar{y} \subseteq ([\bar{x}]_j \cup [\bar{x}]_i)$; and,*
  2. *all distinct proofs w.r.t. j,i up to one-to-one renaming of existential variables (i.e. variables not in $[\bar{x_j}] \cup [\bar{x_i}]$).*

First, we prove that computing this formula is indeed red^n

Lemma 4.1.8. *If $active^n([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ (resp. $active^n([\eta]|_j^{i-1})$) is $red^n([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ (resp. $red^n([\eta]|_j^{i-1})$), then $active^n([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ (resp. $active^n([\eta]|_j^i)$) is $red^n([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ (resp. $red^n([\eta]|_j^i)$).*

Proof: Given the definition of red^n, we only need to prove that $active^n([\eta]|_i$ contains all proofs (up to existential variables renaming) that are in $red^n([\eta]|_i)$. First, if a proof was

in $red^n([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$, by hypothesis it is in $active^n([\eta]|_{i-1}$, so by construction it is still present (up to renaming) in $active^n([\eta]|_i$. Let us consider now a proof that did not exists in $red^n([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$. This means that it uses a literal l from ψ_i. Given the structure of a proof, it must be that all the variables in the proof are in $[\bar{x_i}]$ (resp. $[\bar{x_j}]\cup[\bar{x_i}]$), it follows that all terms are in $active^n([\eta]|_i$ (resp. $active^n([\eta]|^i_j)$), by construction. □

Also:

Lemma 4.1.9. *$active^n$ size is polynomial w.r.t. the number of attributes in the system, considering a fixed database arity.*

Proof: $active^n$ contains either atoms with only free variables (which are at most $2\cdot|\bar{x}|$), or proofs. The possible number of proofs that are distinct, up to one-to-one renaming of existential variables, is polynomial w.r.t. the number of attributes, because in each proof (including at most 2 atoms) there are at most $2\cdot arity$ existential variables, with $arity$ being the maximum arity in the database schema. It follows that the possible proofs are bounded by $O(p\cdot(arity+|x|)^{arity})$. If $arity$ is 1, then the exponent is 2 to consider the contradiction proofs of the form $x\neq y$. □

We now create an automaton that recognizes the set of prefixes s.t. $red^f([\eta]|^{j+n}_j(\langle\psi_i\rangle_{i\leq j+n}))\wedge\bar{x}_j=\bar{x}_{j+n}$ is satisfiable. First, we augment the symbols to the cross products of ther symbolic transitions with $\{-,j,j+n\}$, this serves to identify the j and $j+n$ instants in the string. We build now a non-deterministic automaton $\mathscr{A}$ that performs that accepts only if:

1. there exists a single symbol marked with j, and a single $j+n$ at the end of the string;

2. keeping $active^n([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ in its state up to the symbol $\{\psi,j\}$, and then $active^n([\eta]|^i_j(\langle\psi_k\rangle_{k\leq i}))$, it checks that at the end of the string $active^n([\eta]|^{j+n}_j(\langle\psi_k\rangle_{k\leq j+n}))\wedge[\bar{x_j}]=[\bar{x_{j+n}}]$;

3. it runs $BA_{\neg\varphi}$ on the symbolic prefix, and makes sure that it is on a final state r after the symbol $\{\psi,j\}$ and on the final symbol marked with $j+n$;

The states in $\mathscr{A}$ contain the following information:

- $active^n$;
- the current state of BA;
- flags for cheking existence of j and $j+n$ symbol in the right number and places.

The transition relation maintains $active^n$ by using the Definition 4.1.7, and the state of the BA, by running BA_φ on the symbolic prefix. It is immediate to see that $\mathscr{A}$ checks the conditions of Lemma 4.1.1. It follows that checking emptyness of $\mathscr{A}$ solves the verification problem for artifact systems with no data dependencies and a property φ.

Since the size of $active^n$ depends polynomially w.r.t. the number of attributes, considering a fixed arity database, it follows that we have a number of states that is exponential in the number of attributes. Checking emptyness of a non-deterministic automaton can be performed in LOGSPACE, it follows that:

Lemma 4.1.10. *$\mathscr{A}$ solves the verification problem in PSPACE w.r.t. the number of attributes in the artifact systems, considering a fixed arity database.*

4.2 Key dependencies

In this section we consider an artifact system with a db schema that has unary key dependencies. We assume w.l.o.g. that the schema is normalized, in the sense that if a relation has a key, it only has a single dependent attribute. Also, we assume that the key attribute is always the first. As in the previous section we will create an automaton that checks emptyness of the language of the satisfiable symbolic prefixes s.t. $\bar{x}_j = \bar{x}_{j+n}$ and they satisfy $BA_{\neg\varphi}$. Analogously to the previous case we define the notion of proof. First, however, we have to introduce the following concepts:

Definition 4.2.1. *A* head *is, either:*

1. *a pair $p(\bar{y}) \wedge \neg p(\bar{z})$; or*
2. *an atom $y \neq z$.*

Definition 4.2.2. *A* matching chain *with head* $\{y,z\}$ *is a minimal set of atoms s.t.:*

1. *it contains two atoms* $p(y_k,y)$ *and* $p(z_k,z)$ *with p a predicate with a key constraint; and*

2. *either* $y_k|_{\bar{x}_j \leftarrow \bar{x}_{j+n}} = z_k|_{\bar{x}_j \leftarrow \bar{x}_{j+n}}$*, or it contains a pair of matching chains with head* $\{y_k,z_k\}$.

Definition 4.2.3. *Let* $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ *be in prenex form. We call a* unary-keys contradiction proof *(or simply* proof*) a minimal set of the atoms in* $[\eta]$ *s.t.:*

1. *it contains a head;*

2. *for every corresponding pair of variables* $\{y,z\}$ *in the head (i.e. with* $y \neq z$*, or* $y \in \bar{y}$ *and* $z \in \bar{z}$ *in the same key-attribute position), if* $y|_{\bar{x}_j \leftarrow \bar{x}_{j+n}} \neq z|_{\bar{x}_j \leftarrow \bar{x}_{j+n}}$*, then there is a matching chain with head* $\{y,z\}$.

We will now use the informal notion of derivation rule in the context of a chase. Let $p(x,y) \wedge p(x,z) \rightarrow y = z$ be a key constraint, we say that it can be represented by a derivation rule $p(x,y) \wedge p(x,z) \vdash y = z$. In the same way, the event of a failing chase can be encoded in a series of derivation rules that imply $false$ (or $\perp$). For instance $x \neq x \vdash \perp$.

Lemma 4.2.4. $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ *is satisfiable iff it does not contain any contradiction proof.*

Proof: First, we prove the following claim:

† a matching chain with head $\{y,z\}$ implies $y = z$.

The claim follows from the fact that every link of the chain is the left-side of a key data dependency that results in the equality of the dependent attributes.

Let us assume that $[\eta]$ contains a contradiction proof p. It follows that for every corresponding pair of attributes in the head $\{y,z\}$ is, either equal ($y = z$), or there exists a matching chain, which, by †, implies $y = z$. It follows that the head is a contradiction, hence $[\eta]$ is unsatisfiable.

Now we assume $[\eta]$ is unsatisfiable. It follows that there has to be a way to derive a negation from the terms in $[\eta]$ plus the key dependencies. Remember that $[\eta]$ is in $CQ^{\neg}$ form and we are assuming no constants are present. It follows that the only derivation rules that results in $\bot$ are the following:

1. $p(\bar{x}) \wedge \neg p(\bar{x}) \vdash \bot$;
2. $x \neq x \vdash \bot$.

It follows that any derivation resulting in a contradiction has to end with one of those rules. The only other derivation rules are the key dependencies. It follows that we can build a proof p (as in Definition 4.2.3) by including the terms that are used in the derivation. Since the application of the key derivation rules matches the structure of the matching chains, and the final derivation matches the structure of the head, it is easy to see that there exists a proof p in $[\eta]$. □

In the case with no dependencies, we could simply extract all proofs from $[\eta]$ in order to build red^n. In this case, however, proofs are not bounded in general, and, for feedback-free artifact systems, they are bounded only by the size of $[\eta]$ which is hyperexponential. To overcome this issue, the main idea is to take advantage of the structure of the contradiction proof, by decomposing it in polynomially bounded *fragments* while still maitaining all the infromation necessary to check satisfiability of $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$.

Definition 4.2.5. *We call a* proof fragment *(or* fragment*) f a maximal subset of the terms in a contradiction proof s.t.:*

1. *either f contains a single term $t(\bar{y})$ with $\bar{y} \in \bar{x}_i \cup \bar{x}_k$; or*
2. *for every pair of terms $s(\bar{y})$, $t(\bar{z})$, there exists a sequence of terms $s(\bar{y}) = u_0(\bar{w}_0)\ldots u_n(\bar{w}_n) = t(\bar{z})$ s.t. for every $0 \leq i < n$, $\bar{w}_i \cap \bar{w}_{i+1} \cap (\bar{x}_j \cup \bar{x}_k) \neq \emptyset$ (i.e. they are joined on an existentially quantified variable).*

Note that the above definition guarantees that all joins that are made to create the chaining happen with existentially quantified variables. Also,

Lemma 4.2.6. *In feedback-free systems, the maximum size of a fragment is polynomial w.r.t. the number of attributes in the system, considering a fixed arity database.*

Proof: First, in feedback-free system $[\eta]$ can be rewritten with quantifier depth bounded by $|\bar{x}|^2$ (with $\bar{x}$ being the attributes of the system). Since fragments are ultimately subsets of $[\eta]$, the same property holds. Given the join structure of a chain, it is easy to see that, with a quantifier depth of qd, it is possible to use at most $2 \cdot qd$ existentially quantified variables. Any fragment can at most include a number of matching chains that is bounded by the maximum arity of the database, It follows that any fragments can at most use $O(arity \cdot qd)$ existentially quantified variables. The number of total variables are then $O(arity \cdot qd + |\bar{x}|)$. The number of possible literals in a fragment (i.e. the fragment size) is then bounded by $O(p \cdot (arity \cdot qd + |x|)^{arity})$, with p being the number of predicates in the database. It follows that, if the arity is fixed, the size is polynomial w.r.t. the number of attributes. □

Now, we define the reduced form red^f in a way similar to red^n.

Definition 4.2.7. *Let $\langle \rho_i \rangle_{i \leq j+n}$ be a symbolic run prefix, we define the* key-proof-reduced *inherited constraints $red^f([\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n}))$ in the following way. $red^n([\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n}))$ contains:*

1. *all terms $t(\bar{y})$ s.t. $\bar{y} \subseteq ([\bar{x}]_j \cup [\bar{x}]_i)$; and,*
2. *for each proof fragment f of $[\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n})$, $ren_{red^n([\eta]|_j^{j+n})}(f)$ if there is no one-to-one variable substitution μ of existential veriables s.t. $\mu(ren_{red^n([\eta]|_j^{j+n})}(f)) \subseteq red^n([\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n}))$.*

We can now derive the following result.

Lemma 4.2.8. *$[\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ is satisfiable iff $red^f([\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n})) \wedge \bar{x}_j = \bar{x}_{j+n}$ is satisfiable.*

Proof: We prove the contrapositive: $[\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ is unsatisfiable iff $red^f([\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n})) \wedge \bar{x}_j = \bar{x}_{j+n}$ is unsatisfiable.

Let us assume that $[\eta]|_j^{j+n}(\langle \psi_i \rangle_{i \leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ is unsatisfiable. It follows that it contains contradiction proof p. Also, p can be partitioned in a set of fragments F. Let

us call F^r the renamed versions of the fragment in F (i.e. $F^r = \{ren_{red^n([\eta]|_j^{j+n})}(f) | f \in F\}$). Fragments in F^r appear, up to a rnaming of existentially quantified variables, in $red^f([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})) \wedge \bar{x}_j = \bar{x}_{j+n}$. Let us call p^r the conjunction of the fragments in F^r. We prove now that p^r is syntactically equal to p up to a renaming of existentially quantified variables. Let μ be the composition of the renamings used for the fragments of p to generate the fragments in F^r. The renaming μ is well-formed because, by definition of fragment, no existentially quantified variable appears in more than one fragment of F. It follows that applying μ to p gives exactly p^r. This means that $red^f([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})) \wedge \bar{x}_j = \bar{x}_{j+n}$ is unsatisfiable.

Let us now assume that $red^f([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})) \wedge \bar{x}_j = \bar{x}_{j+n}$ is unsatisfiable. Again, it must contain a proof p^r that can be partitioned in a set of fragments F^r. By construction, these fragments are the renamed versions of a set of fragments F in $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$. The terms in the fragments in F form a contradiction proof in $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$, which is unsatisfiable. □

Given the bound on the size of the fragments it is easy to see that the size of this reduced form is exponential in the number of attributes, given a fixed arity database.

Analogously to the case described in the previous subsection, we can design a precedure that computed red^f while scanning a symbolic prefix. In order to do it we maintain the following formula.

Definition 4.2.9. *We call $active^f([\eta])$, the formula computed while scannning $\langle\psi_i\rangle_{i\leq j+n}$ in the following way:*

- $active^f([\eta](\langle\psi_0\rangle)) = [\eta](\langle\psi_0\rangle)$*;*
- *we keep track of the equivalence classes using the equality atoms in the symbolic transitions;*
- *for all $i <= j$, we build $active^f([\eta]|_i)$ from $active^f([\eta]|_{i-1,i}(\langle\psi_k\rangle_{k\leq i}))$ (i.e. $[\eta](\langle\psi_k\rangle_{k\leq i})$ with existential quantifiers for $[\bar{x}]_k$, $k <= i-2$) keeping only:*
  1. *the terms $t(\bar{y})$ s.t. $\bar{y} \subseteq [\bar{x}]_i$; and*
  2. *for every fragment f in $active^f([\eta]|_{i-1}) \wedge \psi_i$ w.r.t. free variables $[\bar{x}]_i$, we keep a renamed fragment $ren_{active^f([\eta]|_{i-1})\wedge\psi_i}(f)$;*

- *for all $j < i <= j+n$, we build $active^f([\eta]|_j^i)$ from $active^f([\eta]|_j^{i-1,i}(\langle\psi_k\rangle_{k\leq i}))$ keeping only:*

  1. *all terms $t(\bar{y})$ s.t. $\bar{y} \subseteq ([\bar{x}]_j \cup [\bar{x}]_i)$; and,*
  2. *for every fragment f in $active^f([\eta]|_j^{i-1}) \wedge \psi_i$ w.r.t. free variables $[\bar{x}]_j \cup [\bar{x}]_i$ we keep a renamed fragment $ren_{active^f([\eta]|_j^{i-1})\wedge\psi_i}(f)$;*

The soundness of the above procedure is proved by the following lemma.

Lemma 4.2.10. *If $active^f([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ (resp. $active^f([\eta]|_j^{i-1})$) is $red^f([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ (resp. $red^f([\eta]|_j^{i-1})$) (up to one-to-one existential variables renaming), then $active^f([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ (resp. $active^f([\eta]|_j^i)$) is $red^f([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ (resp. $red^f([\eta]|_j^i)$)*

Proof: Let f be a fragment in $red^f([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$. It follows that, if it contains only terms with variables in $[\bar{x}]_{i-1} \cup [\bar{x}]_i$, then f is in $active^f([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ because those terms appear in $active^f([\eta]|_{i-1,i}(\langle\psi_k\rangle_{k\leq i}))$. Let us analyze the case when f contains some terms with variables not in $[\bar{x}]_{i-1} \cup [\bar{x}]_i$, let us call the set of those term T. Since all the terms in T are part of a fragment f, they are part of a contradiction proof. It follows that every term in T was part of a proof in $red^f([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ and so appeared in $active^f([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ in some fragment. Note that different terms in T might have appeared in different fragments of $active^f([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$, we call F the set of these fragments. Also, by construction, terms in T that come from different fragments of $active^f([\eta]|_{i-1}(\langle\psi_k\rangle_{k\leq i-1}))$ can only be joined in f through terms in ψ_i. Since $active^f([\eta]|_{i-1,i}(\langle\psi_k\rangle_{k\leq i}))$ contains those terms, along with all the fragments F, it must contain f, which then appears in $active^f([\eta]|_i)$.

The case for the respective formulas is analogous. □

Note that, since the size of the fragments is polynomial w.r.t. the number of attributes, it follows that, since $active^f$ can at most include all possible fragments, its size is bounded by an exponential function w.r.t. the number of attributes, considering a fixed arity database.

Analogously to the previous section, it is easy to see that we can build an automaton that recognizes the language of symbolic prefixes s.t. $red^f([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})) \wedge$

$\bar{x}_j = \bar{x}_{j+n}$. Again, we augment the symbols to the cross products of ther symbolic transitions with $\{-, j, j+n\}$, this serves to identify the j and $j+n$ instants in the string. We build now a non-deterministic automaton $\mathscr{A}$ that performs that accepts only if:

1. there exists a single symbol marked with j, and a single $j+n$ at the end of the string;

2. keeping $active^f([\eta]|_i(\langle\psi_k\rangle_{k\leq i}))$ in its state up to the symbol $\{\psi, j\}$, and then $active^f([\eta]|_j^i(\langle\psi_k\rangle_{k\leq i}))$, it checks that at the end of the string $active^f([\eta]|_j^{j+n}(\langle\psi_k\rangle_{k\leq j+n})) \wedge [\bar{x_j}] = [\bar{x_{j+n}}]$;

3. it runs $BA_{\neg\varphi}$ on the symbolic prefix, and makes sure that it is on a final state r after the symbol $\{\psi, j\}$ and on the final symbol marked with $j+n$;

The states in $\mathscr{A}$ contain the following information:

- $active^f$;
- the current state of BA;
- flags for cheking existence of j and $j+n$ symbol in the right number and places.

The transition relation maintains $active^f$ by using the Definition 4.2.9, and the state of the BA, by running BA_φ on the symbolic prefix. It is immediate to see that $\mathscr{A}$ checks the conditions of Lemma 4.1.1. It follows that checking emptyness of $\mathscr{A}$ solves the verification problem an artifact system with unary keys and property φ.

Since the size of $active^f$ depends exponentially w.r.t. the number of attributes, considering a fixed arity database, it follows that we have a number of states that is doubly exponential in the number of attributes. Checking emptyness of a non-deterministic automaton can be performed in LOGSPACE, it follows that:

Proposition 4.2.11. *$\mathscr{A}$ solves the verification problem in EXPSPACE w.r.t. the number of attributes in the artifact systems, considering a fixed arity database.*

4.2.1 Navigational complexity

In this subsection we introduce the notion of navigational complexity, which helps to better understand the complexity of the technique described above. First, we call *chain* a conjunction of positive literals s.t. $p(x,y)$ is a chain rooted in x if p is a predicate with a key constraint; and $p(w,z) \wedge c(z)$ if p is a predicate with a key and $c(z)$ is a chain rooted in z.

Definition 4.2.12. *The navigational complexity of an artifact system-property pair* (Γ, φ_f) *is the maximum length of a chain that appears in any rewritten inherited constraint* $[\eta^*]$ *of* (Γ, φ_f).

Intuitively, the length of the chains represent to what extent the business process navigates the key dependencies in the database. Clearly, by Lemma 4.2.6, in feedback-free systems the maximum navigational complexity is polynomially bounded by the number of attributes in the system. However, we argue that navigational complexity usually depends on the structure of the database schema and business processes are not used to perform extensive navigation. It is then interesting to explore the complexity in the case of constant navigational complexity.

Proposition 4.2.13. *Considering constant navigational complexity, verifying a property* φ *on a feedback-free artifact system using unary key dependencies and no arithmetic is in PSPACE.*

Proof: In the case of constant navigational complexity the number of possible fragments is polynomially bounded. This follows from the fact that the maximum number of terms s in a contradiction proof is bounded by $2 \cdot arity \cdot (nav + 1)$, where nav is the navigational complexity. It follows that the maximum number of existential variables $|V|$ is bounded by the above size times the arity. The total number of variables is then $|V| + |\bar{x}|$, where $\bar{x}$ are the attributes of the system. The possible terms are then $2 \cdot p \cdot (|X| + |\bar{x}|)^{arity}$. Since fragments are subsets of proofs, the number of possible fragments is $O((2 \cdot p \cdot (|X| + |\bar{x}|)^{arity})^s)$, which is polynomial in the number of variables in the system $|\bar{x}|$.

The verification complexity upper bound comes from the above and Proposition 4.2.11. □

A sufficient condition can be derived from the database schema by imposing a specific form on the pre and post-conditions of services. Intuitively, we want to characterize the common class of business processes that use joins just to navigate foreign keys. If this is the case, when the foreign key schema is acyclic, we have a navigational complexity that depends on the database schema.

Formally, we define a graph $E_{(\Gamma,\varphi)}$ from the services of Γ and a property φ, that has one node for each attribute and an edge (a,b) if there exists an equality $a = b$ (or $a' = b'$, $a = b'$,$a' = b$) in any pre or post condition in the services of Γ or in φ.

Definition 4.2.14. *Assuming a schema with foreign key dependencies, we call* foreign-key-navigation system *an artifact system where in any pre or post-condition of any service, for every attribute variable a that appears in a term $t(a,b)$ as a key attribute, all the variables r reachable from a in E_Γ (including a), when they appear in a database term $s(c,a)$, there is a foreign key constraint $s_2 \subseteq t_1$ in the schema.*

It is immediate that the above definition guarantees, that in any computational graph of (Γ,φ) there is no join on a key attribute that is not encoded in the foreign key constraints. It follows:

Lemma 4.2.15. *With a database schema DB with acyclic foreign-keys, a foreign-key-navigation system has constant navigational complexity w.r.t. the number of attributes, given a fixed database schema.*

Proof: Since all the joins in the inherited constraints that involve key attributes follow the foreign keys dependencies, it follows that the maximum length of a chain is bounded (as the foreign keys are acyclic) and it depends only on the size of the database schema. It follows that with a fixed database schema, we have a navigational complexity that is constant. □

From Proposition 4.2.13.

Corollary 4.2.16. *The verification problem for a foreign-key-navigation system (Γ,φ) is PSPACE in the size of the system, given a fixed database schema.*

4.2.2 Relaxing feedback-freedom

In the technique we just presented we use the feedback free defintion only to bound the size of the fragments. Note, however, that the feedback-freedom as it is presented in Definition 2.3.2, is more restrictive than what we require. We now present a modified version that is more lenient.

In order to state our definition we change the defintion of computational graph by modifying the G_ψ graph associated to the symbolic transitions. In Section 2.3, G_ψ was defined as the restriction to $\bar{x}, \bar{x}'$ of the transitive closure of the graph containing an edge among every two variables occurring together in an atom of ψ. We define, now G^k_ψ as the restriction to $\bar{x}, \bar{x}'$ of the transitive closure of the graph containing an edge for every two variables x_i and x_j occurring in an atom $p(x_i, x_j)$ of ψ, s.t. p has a key constraint. Let ρ be a symbolic run, G^k_ρ is defined from G^k_ψ as in Section 2.3.

Definition 4.2.17. (Γ, φ_f) *is* key-feedback-free *iff for every symbolic run prefix* $\rho = \{\psi_i\}_{i \leq n}$*, each path from* x_i *to* x_j *in* G^k_ρ *contains a node* y *such that* $span([x_i]) \cup span([x_j]) \subseteq span([y])$.

Note that this definition does not imply anymore that the inherited constraints can be rewritten with bounded quantifier depth. However, since, matching chains are connected components that contain only predicates with keys, they can be rewritten with bounded quantifier depth. It follows from Lemma 4.2.6:

Corollary 4.2.18. *In key-feedback-free systems, the maximum size of a fragment is polynomial w.r.t. size of system.*

So we have:

Corollary 4.2.19. *Verifying a property* φ *on a key-feedback-free artifact system using unary key dependencies is in EXPSPACE (PSPACE with constant navigational complexity)*

4.3 (Re-)Introducing arithmetic

The reduced form of inherited constraints introduced in Section 3.1.1 has a size that is hyperexponential in the number of attributes, even when no dependencies or

arithmetic is present. In the rest of this chapter we proved how, in the absence of arithmetic, we are able to discard information from the inherited constraints to a much more manageable level. Also, in Lemma 3.2.3 we proved that we can effectively separating the satisfiability of the database portion of the inherited constraints from the arithmetic portion by guessing equality types of common variables. The main idea of this section is then to define a reduced form that does not incur in the hyperexponential size unless arithmetic is present.

First, we denote with $[\eta_{db}]$ the subset of terms of $[\eta]$ that either refer to database predicates or to equalities and inequalities. Analogously, we denote with $[\eta_a]$ the subset with only equalities, inequalities and arithmetic terms.

As clear from Lemma 3.2.3, the satisfiability of $[\eta]$ depends on the equality type of the common variables. In order to define a reduced form that takes this into account, we call $A_{[\eta_a]}$ the set of variables appearing in $[\eta_a]$. Since we want to maitain all information regarding the equality types of this variables, we define the following reduced form.

We denote with $[\eta]|_V(\langle\psi_i\rangle_{i<k})$ the formula $\exists\bar{z}([\eta](\langle\psi_i\rangle_{i<k}))$, where $\bar{z} = \{[\bar{x}]_i | i < k\} \setminus V$.

Definition 4.3.1. *Let $\langle\rho_i\rangle_{i\leq j+n}$ be a symbolic run prefix, we define the* mixed-reduced *inherited constraints $red^m([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ the formula $\exists\bar{z}(red^f([\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n}))) \wedge [\eta_a]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})$, where $V = \bar{x}_j \cup \bar{x}_{j+n} \cup A_{[\eta_a]}$ and $\bar{z} = V \setminus (\bar{x}_j \cup \bar{x}_{j+n})$.*

Note how the construction of the fragments in $red^f([\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n}))$ changes as a result of the different quantification. We have then:

Lemma 4.3.2. *Let $eq(V)$ be an equality type of the variables in V. $red^m([\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n})) \wedge eq(V)$ is satisfiable iff $[\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n}) \wedge eq(V)$ is satisfiable.*

Proof: We prove the contrapositive: $red^m([\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n})) \wedge eq(V)$ is unsatisfiable iff $[\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n}) \wedge eq(V)$ is unsatisfiable.

Let us assume that $[\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n}) \wedge eq(V)$ is unsatisfiable. It follows that it contains contradiction proof p. Also, p can be partitioned in a set of fragments F and

of atoms in $eq(V)$. Let us call F^r the renamed versions of the fragment in F. Fragments in F^r appear, by construction, in $red^f([\eta_{db}]|_V(\langle\psi_i\rangle_{i\leq j+n}))\wedge eq(V)$. Let us call p^r the conjunction of all the fragments in F^r and of $eq(V)$. We prove now that p^r is syntactically equal to p up to a renaming of existentially quantified variables. Let μ be the composition of the renamings used for the fragments of p to generate the fragments in F^r. The renaming μ is well-formed because, by definition of fragment, no existentially quantified variable appears in more than one fragment of F. It follows that applying μ to p gives exactly p^r, because p is formed only by fragments and $eq(V)$ and $eq(V)$ contains no existential quantified variable. This means that $red^f([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))\wedge\bar{x}_j=\bar{x}_{j+n}$ is unsatisfiable.

Let us now assume that $red^f([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))\wedge\bar{x}_j=\bar{x}_{j+n}$ is unsatisfiable. Again, it must contain a proof p^r that can be partitioned in a set of fragments F^r plus $eq(V)$. By construction, the fragments in F^r are the renamed versions of a set of fragments F in $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})\wedge\bar{x}_j=\bar{x}_{j+n}$. The terms in the fragments in F plus $eq(V)$ form then a contradiction proof in $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})\wedge\bar{x}_j=\bar{x}_{j+n}$, which is unsatisfiable. □

From Lemma 3.2.3 and Lemma 4.3.2, it follows:

Corollary 4.3.3. *$[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})\wedge\bar{x}_j=\bar{x}_{j+n}$ is satisfiable iff $red^m([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))\wedge\bar{x}_j=\bar{x}_{j+n}$ is satisfiable.*

4.3.1 Incrementally maintaining $red^m([\eta])$

We now present the incremental procedure to maitain red^m. The main idea is maintain separately the database and the arithmetic portions. The maintenance of the database portion depends on the arithmetic part, so we start by describing that one.

First, we define the following concept:

Definition 4.3.4. *We call a* general fragment *f a subset of the terms of $[\eta]|_j^k$ s.t.:*

1. *either f contains a single term $t(\bar{y})$ with $\bar{y}\in\bar{x}_i\cup\bar{x}_k$; or*
2. *for every pair of terms $s(\bar{y})$, $t(\bar{z})$, there exists a sequence of terms $s(\bar{y})=u_0(\bar{w}_0)\ldots u_n(\bar{w}_n)=t(\bar{z})$ s.t. for every $0\leq i<n$, $\bar{w}_i\cap\bar{w}_{i+1}\cap(\bar{x}_j\cup\bar{x}_k)\neq\emptyset$ (i.e. they are joined on an existentially quantified variable).*

Now we outline the incremental maintenance procedure:

- $red([\eta_a](\psi_0)) = [\eta_a](\psi_0)$;
- we keep track of the equivalence classes using the equivalence atoms in the symbolic transitions ψ_i;
- for all $i <= j$, we build $red([\eta_a]|_i)$ from $red([\eta_a]|_{i-1,i}(\langle\psi_k\rangle_{k\leq i}))$ (i.e. $[\eta](\langle\psi_k\rangle_{k\leq i})$ with existential quantifiers for $[\bar{x}]_k$, $k <= i-2$) keeping only:
  1. the terms $t(\bar{y})$ s.t. $\bar{y} \subseteq [\bar{x}]_i$; and
  2. for every general fragment f in $red([\eta_a]|_{i-1}) \wedge \psi_i$ w.r.t. free variables $[\bar{x}]_i$, we keep a renamed fragment f^r;
- for all $j < i <= j+n$, we build $red([\eta_a]|^i_j)$ from $red([\eta_a]|^{i-1,i}_j(\langle\psi_k\rangle_{k\leq i}))$ keeping only:
  1. all terms $t(\bar{y})$ s.t. $\bar{y} \subseteq ([\bar{x}]_j \cup [\bar{x}]_i)$; and,
  2. for every fragment f in $red([\eta_a]|^{i-1}_j) \wedge \psi_i$ w.r.t. free variables $[\bar{x}]_j \cup [\bar{x}]_i$ we keep a renamed fragment f^r;

Note the difference between this procedure and the one presented in Section 4.2: the above one uses general fragments, which are not subsets of a contradiction proof. We have, then:

Lemma 4.3.5. *Let us denote by $P(red([\eta_a](\langle\psi_i\rangle_{i\leq k})), \psi_{k+1})$ the application of the above procedure to the reduced form in instant k and the symbolic transition ψ_{k+1}, then $P(red([\eta_a](\langle\psi_i\rangle_{i\leq k})), \psi_{k+1}) = red([\eta_a](\langle\psi_i\rangle_{i\leq k+1})$ (resp. $P(red([\eta_a]^k_j(\langle\psi_i\rangle_{i\leq k})), \psi_{k+1}) = red([\eta_a]|^{k+1}_j(\langle\psi_i\rangle_{i\leq k+1}))$.*

Proof: First, we elaborate on how we keep track of the equivalence classes while extending the inherited constraints with ψ_{k+1}. We create a new existential quantifier for all new equivalence class variables, and we close the scope of all equivalence classes $[e]$ s.t. no $x \in \bar{x}_{k+1}$ is in $[e]$. We also note that, by definition of inherited constraints, $[\eta_a](\langle\psi_i\rangle_{i\leq k+1})$ is the same as $\exists\bar{z}([\eta_a](\langle\psi_i\rangle_{i\leq k}) \wedge [\psi_{k+1}])$, with

$\bar{z}$ the set of equivalence class variables s.t. no $x \in \bar{x}_{k+1}$ appears in $[z] \in \bar{z}$. We prove now that $\exists\bar{z}(red([\eta_a](\langle\psi_i\rangle_{i\leq k})) \wedge [\psi_{k+1}]) \sim [\eta_a](\langle\psi_i\rangle_{i\leq k+1})$. First, we know, by definition of *red* that $red([\eta_a](\langle\psi_i\rangle_{i\leq k})) \sim [\eta_a](\langle\psi_i\rangle_{i\leq k})$. This means that all subtrees in $\mathscr{T}(red([\eta_a](\langle\psi_i\rangle_{i\leq k})))$ appear in $\mathscr{T}([\eta_a](\langle\psi_i\rangle_{i\leq k}))$ and vice versa. It follows, by construction, that all subtrees in $\mathscr{T}(\exists\bar{z}(red([\eta_a](\langle\psi_i\rangle_{i\leq k})) \wedge [\psi_{k+1}]))$ either come from $\mathscr{T}([\eta_a](\langle\psi_i\rangle_{i\leq k}))$ or are trees from $\mathscr{T}(\exists\bar{z}([\psi_{k+1}]))$ with trees from $\mathscr{T}([\eta_a](\langle\psi_i\rangle_{i\leq k}))$ added as leaves. Remember that, by definition of inherited constraints, $[\eta_a](\langle\psi_i\rangle_{i\leq k+1}) = \exists\bar{z}([\eta_a](\langle\psi_i\rangle_{i\leq k}) \wedge [\psi_{k+1}])$. It follows that we can apply the same reasoning as above in the opposite direction. This means that all trees in $[\eta_a](\langle\psi_i\rangle_{i\leq k+1})$ come either from $[\eta_a](\langle\psi_i\rangle_{i\leq k})$ (which menas also from $red([\eta_a](\langle\psi_i\rangle_{i\leq k}))$) or from combining trees from $[\eta_a](\langle\psi_i\rangle_{i\leq k})$ as leaves of trees from $\mathscr{T}(\exists\bar{z}([\psi_{k+1}]))$. Let us call P^p a modified version of the incremental procedure above that does not remove duplicate fragments w.r.t. renaming. It follows that $P^p(red([\eta_a](\langle\psi_i\rangle_{i\leq k})), \psi_{k+1}) = \exists\bar{z}(red([\eta_a](\langle\psi_i\rangle_{i\leq k})) \wedge [\psi_{k+1}]) \sim [\eta_a](\langle\psi_i\rangle_{i\leq k+1})$.

We now have to prove that $P(red([\eta_a](\langle\psi_i\rangle_{i\leq k})), \psi_{k+1})$ is the reduced version of $P^p(red([\eta_a](\langle\psi_i\rangle_{i\leq k})), \psi_{k+1})$ First, we note that the definition of general fragment corresponds to the one of top-level subtrees in $\mathscr{T}([\eta_a](\langle\psi_i\rangle_{i\leq k+1}))$. It follows that we can easily prove the following invariant:

‡ the procedure P output never contains a duplicated top-level tree.

This follows immediately from the fact that the output contains only general fragments that are distinct w.r.t. existental variable renamings. Let us consider now the top-level trees T of $\mathscr{T}(P(red([\eta_a](\langle\psi_i\rangle_{i\leq k}), \psi_{k+1})$ that were not top-level trees in $\mathscr{T}(red([\eta_a](\langle\psi_i\rangle_{i\leq k})))$. By construction, the children of trees in T were top-level trees in $\mathscr{T}(red([\eta_a](\langle\psi_i\rangle_{i\leq k})))$ or come from $\mathscr{T}(\exists\bar{z}[\psi_{k+1}])$. It follows that there are no duplicates in the children of the top-level trees in $\mathscr{T}(P(red([\eta_a](\langle\psi_i\rangle_{i\leq k}), \psi_{k+1})$. Moreover, by ‡ we know that there are no duplicates in top-level trees. It follows that $P(red([\eta_a](\langle\psi_i\rangle_{i\leq k})), \psi_{k+1})$ is a well-formed reduced form.

The respective case is analogous. □

Since we can incrementally maintain the arithmetic portion of red^m, it follows that we can easily modify the procedure in Section 4.2 to incrementally maintain the

database portion (since the free variables contain the current variables $\bar{x}_i$ the proof is substantially unaltered).

4.3.2 Complexity

In the mixed reduced form, the number of possible fragments depend on the number of variables in the arithmetic part.

Lemma 4.3.6. *For feedback-free systems, the size of* $red^m([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ *is* $O(h(w_a^2))$*, for some* $h \in Hyp$*, with* w_a *the width of the maximum connected component with only arithmetic terms.*

Proof: The size of the arithmetic portion is $O(h(w_a^2))$ (Lemma 3.1.8). From the same counting reasoning in Lemma 4.2.6 we derive the size of the database portion to be in $O(h(w_a^2))$. □

With this result we have a verification procedure whose complexity is in EXPSPACE (PSPACE with constant navigational complexity) when no arithmetic is used and hyperexponential in the width of the largest connected component in $[\eta_a]$ (from Lemma 4.3.6 when arithmetic is used.

4.3.3 Relaxing feedback freedom

As in the previous section, we are able to relax the feedback freedom definition.

In order to state our definition we change the defintion of computational graph by modifying the G_ψ graph associated to the symbolic transitions. Let ψ_{db} be the subset of ψ whose term refer only to database predicates, equalities or inequalities. Let ψ_a be the subset of ψ whose term refer only to arithmetic predicates, equalities or inequalities. Let us define $G^k_{\psi_{db}}$ as is Section 4.2. Let ρ be a symbolic run, G^k_ρ is defined from $G^k_{\psi_{db}}$ as in Section 2.3. Analogously, we define G^a_ρ from G_{ψ_a}. The intuition of this definition is that we account for database predicates and arithmetic predicates separately.

Definition 4.3.7. (Γ, φ_f) *is* arithmetic-key-feedback-free *iff for every symbolic run prefix* $\rho = \{\psi_i\}_{i\leq n}$*, each path from* x_i *to* x_j *in both* G^k_ρ *and* G^a_ρ *contains a node* y *such that* $span([x_i]) \cup span([x_j]) \subseteq span([y])$.

In order to use the above definition in verification we have to prove two results. The first is:

Lemma 4.3.8. *In an artihmetic-key-feedback-free artifact system and property with key constraints and arithmetic, there exists a run ρ satisfying $\neg\varphi$ iff*

1. *there exists a symbolic run prefix $\langle\psi_i\rangle_{i<j+n}$ s.t. $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ is satisfiable, and*

2. *for such $\langle\psi_i\rangle_{i<j+n}$, there exists a run $\{q_i\}_{i\leq j+n}$ of $B_{\neg\varphi}$ on $\{\sigma_i\}_{i\leq j+n}$ (i.e. the run of truth assignments for FO components in $\neg\varphi$ given the run $\{\psi_i\}_{i<j+n}$) such that for some accepting state r. $q_j = q_{j+n} = r$.*

Proof: The if portion is immediate. Let us consider the only if. Let $\langle\psi_i\rangle_{i\leq j+n}$ be a symbolic run that satisfies ρ. The idea is to consider separately the values in ρ that in $\langle\psi_i\rangle_{i\leq j+n}$ appear only in database predicates and those that appear in at least an arithmetic one. By Lemma 4.1.1 we can identify two instants j and i_e such that $[\eta_{db}]|_j^{i_e}(\langle\psi_i\rangle_{i\leq i_e}) \wedge \bar{x}_j = \bar{x}_{i_e}$ is satisfiable, where $[\eta_{db}]$ is the subset of $[\eta]$ that excludes all arithmetic terms. By modifying the technique in Lemma 4.1.1, and knowing that the number of possible $red([\eta_a])$ is finite, we can identify two instants j' and i'_e s.t. $[\eta_{db}]|_{j'}^{i'_e}(\langle\psi_i\rangle_{i\leq i_e}) \wedge \bar{x}_j = \bar{x}_{i_e}$ is satisfiable. Also, we know that $[\eta_a]$ is associated to a feedback-free computation graph G^a_ρ. It follows that we can apply the same technique in Lemma 3.1.10 to prove that there is a way to modify the assignments of the values in the connected components intersecting instant i'_e s.t. $\bar{x}_{j'} = \bar{x}_{i'_e}$. The thesis follows. □

The second result we need is the boundedness of the reduced form that we use.

Lemma 4.3.9. *In artihmetic-key-feedback-free artifact systems the size of $red^m([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}))$ is $O(h(w_a^2))$, for some $h \in Hyp$, with w_a the width of the maximum connected component in G^a_ρ.*

Proof: Note that the size of the arithmetic portion depends only on the connected components in G^a_ρ, it follows that since it is feedback-free the same bound as in feedback-free systems applies ($O(h(w_a^2))$). Also, the size of the database portion depends on the number of variables in the arithmetic portion (which is unchanged compared to feedback free

systems) and on the lenght of the chains in the fragments. Since chains are connected components in G^k_ρ, which is feedback-free, we have the same polynomial bound w.r.t. the number of variables (which is bounded by $O(h(w_a^2))$). It follows that the we have an $O(h(w_a^2))$ for the database portion too. □

We have then:

Corollary 4.3.10. *Verifying a property φ on a arithmetic-key-feedback-free artifact system using unary key dependencies is in $O(h(w_a^2))$, for some $h \in Hyp$, with w_a the width of the maximum connected component in G^a_ρ.*

4.4 Constants

The introduction of constants does not change the above results and complexity bounds for the following reason. The only difference that constants introduce is that additional forms of contradictions are possible. It follows that, in the above development, we have to modify the definitions of contradiction proofs (Definition 4.1.2 and 4.2.3) in order to consider constants. The rest of the theory remains the same, and it is easy to see that the inclusion of constants does not change the size of fragments, and consequently the worst-case complexity of the algorithms. The additional contradictions arising from constants are simply:

1. $c_1 = c_2$, with c_1 and c_2 constants;
2. $c \neq c$, with c a constant.

5 Acyclic Workflows with Exceptions

To analyze the feasibility of the approach described so far, we had to create a model at a much higher level than the one based on services provided in Chapter 2. The main requirements of the model are:

- it has to be based on common usage patterns in business process specification;
- it has to be amenable to randomized generation;
- it has to naturally provide a restriction guaranteeing feedback freedom.

In order to describe this high-level model, called *Acyclic Workflows with Exceptions* (AWE), we will divide its description in a part related to the workflow and in one related to the data conditions. We made this decision to better anchor our decisions to the research literature of the field. We will then define the formal semantics of our model by providing a translation procedure to the model presented in Chapter 2. We will also describe how this high-level model naturally relates to the feedback freedom restriction.

5.1 AWE syntactic model

As it is customary [VDATHKB03], we model a workflow as a way to control the execution of a series of work items. We derive our model from a series of studies aimed at identifying the most common patterns in workflow specifications and in commercial workflow execution engines [VDATHKB03, RtHEvdA05, RHE05, RvdAtH, TLR07]. In order to fully evaluate the power of the verification techinque that we present in this work, we introduce two extra elements compared to the cited works: artifact attributes and an external read-only database. In this section we describe our workflow model

starting from the basic building block (the work item, or *basic activity*) and then introducing the various construct used to control their execution. The formal syntax is described in Section 5.2 after all the elements of the models have been informally introduced.

5.1.1 Basic activities

Following the models for work items in [RHE05, RvdAtH], a basic activity has a little internal workflow that tracks its execution status. They also have a precondition that enables them to start. This precondition depends on the workflow structure. We model many kinds of activities. One distinction is between manual and automatic activities. Automatic activities have the the following states {`created`, `started`, `completed`, `failed`}. All activities start in the `created` state. Then, if the precondition is satisfied (e.g. the previous activity is completed), they transition to the `started` state, which can result into a `completed` activity or a `failed` one. Manual activities have an additionial state called `offered`. A manual activity transition from `created` to `offered` as it has to be assigned to a human performer (our model does not model the different assignment strategies described in [RHE05]). When assigned, the activity transitions to the the `started` state, which, again, can result into a `completed` activity or a `failed` one. The state of each activity is stored in an artifact attribute called 'status attribute'.

In addition to the manual/automatic distinction, an activity might also modify some artifact data attributes, called 'output attributes' of an activity. Note that we assume the output attributes of all activities to be pairwise disjoint. In case of an activity that modifies data, it does so when transitioning to the `completed` state. The data values of a set of 'output attributes' of an activity is set through a postcondition using data attributes and status attributes and making use of database predicates and arithmetic operations as in service postconditions in Chapter 2.

Definition 5.1.1. *A* basic activity *is a tuple* $\{type, O, \psi, exc\_hnd\}$*, where type can be either automatic or manual, O is a set of output attributes,* ψ *is a postcondition referring to output artifact attributes as primed variables , and exc_hnd is an exception handling policy.*

Exception handling policies will be introduced in Subsection **??**. The intuitive semantics is stated above, and the formal semantics will be given in Section 5.2 with a reduction to the service-based model of Chapter 2.

5.1.2 Sub-workflow

A sub-workflow (or subflow) is a composite activity formed by a sequence of activities. Sub-workflows form a hierarchy that is exploited for exception handling (Subsection 5.1.4) and variable scopes of data conditions (Subsection 5.3). This use of the hierarchy of workflows is present in real world patterns as presented in [RvdAtH] and [RtHEvdA05]. Besides the hierarchy, a subflow is simply a control structure that forces activities (either basic or composite) to be performed in a fixed order. When the precondition for a subflow is satisfied, every activity has to wait for its predecessor to be completed (i.e. let a be an activity and p_a its predecessor in the subflow sequence, the the precondition for a simply states that p_a has to be in state `completed`). The subflow is completed when the last activities reaches the state `completed`.

Definition 5.1.2. *A* sub-workflow *is a tuple* $\{\sigma, exc\_hnd\}$ *where* σ *is a sequence of activities and exc_hnd is an exception handling policy.*

Also, every workflow is defined as being the root subworkflow of the above mentioned hierarchy.

5.1.3 Splits

We have two kinds of composite activities in our model that split the flow of execution: AND-Splits and XOR-Splits. AND-Splits can have any number of branches. Each branch is a sequence of activities (like a subflow). Branches in an AND-Split execute in parallel and in any possible interleaving order. The AND-Split is considered complete when all branches are completed.

Definition 5.1.3. *An* AND-Split *is a unary tuple* $\{B\}$*, where B is a set of sequences of activities.*

XOR-Splits have two branches, and are associated to a logical condition on data and status attributes, that can refer to database predicates and arithmetic operations. The two branches are associated to a possible evaluation of the condition (either true or false). A branch precondition is the conjunction of the precondition of the XOR-Split with the associated evaluation of the XOR-Split condition. The execution of the branches in a XOR-Split is exclusive: unrespective of the XOR condition, a branch cannot start executing if the other branch is already started. A XOR-Split is considered completed when the executing branch is completed.

Definition 5.1.4. *A* XOR-Split *is a tuple* $\{b_{true}, b_{false}, \chi\}$ *where* b_{true} *and* b_{false} *are two sequences of activities, and* χ *is a condition on artifact attributes.*

5.1.4 Exceptions

Exceptions in our model have a very important role. As proposed in [AM00] and implemented in an execution engine [DBL01], in our model exceptions are the only way to generate cycles in the workflow. This assumption is especially relevant when paired with the considerations that led to the development of the feedback freedom class of business processes.

The semantics of exceptions in AWEs, is based on the idea of exception handling policies. These policies specify a way to react to the fact that an activity fails. We already saw how basic activities can fail.

We adopted the most commonly supported exception handling patterns from [RvdAtH] and provided a way to choose between these policies in the workflow activities. The possible policies are:

1. *restart*: if the activity fails, it is restarted (i.e. put in state `started`);
2. *reallocate*: if a manual activity fails, it is offered to potentially another humen performer (i.e. put in state `offered`);
3. *fail*: the parent sub-flow (or the whole business process) fails.

Note that, for basic activities, the only thing affected by the restart or reallocation is its status attribute. In the case, however, that a whole subflow fails, the restart handling

conditions resets all statuses of the contained activities to `created` and possibly also their output attributes. This implicitly creates a 'cycle' in the workflow (i.e. some completed activities will have to be completed again, possibly with different outcomes). Many other handling patterns are described in [RvdAtH], we choose however the most commonly implemented in workflow execution engines.

5.2 Semantics of AWEs

In this section, we define the semantics of AWEs by reducing it to the lower-level service-based model of Chapter 2.

5.2.1 Attributes

Assuming a database schema, the first thing we have to define are the artifact attributes. From Subsection 5.1.1 it should be clear that, for each basic activity a there is a status attribute (that we call $status_a$) and a set of output attributes (O_a). Since an unhandled exception can possibly result in te failure of the whole business process, we decide to explicitly store the status of the whole business process in an attribute (`global_status`) that can take the values {`started`, `completed`, `failed`}. Remember that the output attributes of basic activities are pairwise disjoint. It follows that the set of attributes $\mathscr{A}$ is:

$$\bigcup_{a \in \mathscr{B}} (status_a \cup O_a) \cup \{\texttt{global\_status}\}$$

with $\mathscr{B}$ being the set of all basic activities.

5.2.2 Basic activities services

The evolution of the workflow is specified by the changes in the the basic activities. It follows that we have a service for each such transitions. In the hierarchical structure of the workflow, every basic activity has a composite activity that controls it. The controlling composite activity (e.g. a subflow or a XOR-Split) implicitly provides a precondition to the basic activity. The basic activity is allowed to leave the `created`

state only when this precondition is true. For instance, the precondition of the first activity of a XOR branch has to include the branch condition. We will describe how the composite activities assign precondition later. Now we show the services associated to an automatic basic activity a. Let $P(A)$ be the following formula depending on a set of attributes A:

$$\bigwedge_{a \in A} (a' = a).$$

For each basic activity we have:

1. A service $created-to-started_a$ with precondition $\pi_a \wedge status_a = \text{"}\texttt{created}\text{"}$ and postcondition $status'_a = \text{"}\texttt{started}\text{"} \wedge P(\mathscr{A} \setminus \{status_a\})$;

2. A service $started-to-completed_a$ with precondition $status_a = \text{"}\texttt{started}\text{"}$ and postcondition $status'_a = \text{"}\texttt{completed}\text{"} \wedge P(\mathscr{A} \setminus (\{status_a\} \cup O_a)) \wedge \psi_a$, with ψ_a being the postcondition on the output attributes O_a;

3. A service $started-to-failed_a$ with precondition $status_a = \text{"}\texttt{started}\text{"}$ and postcondition $status'_a = \text{"}\texttt{failed}\text{"} \wedge P(\mathscr{A} \setminus \{status_a\})$;

For manual activities there is an extra state between `created` and `started`, so the services are:

1. A service $created-to-offered_a$ with precondition $\pi_a \wedge status_a = \text{"}\texttt{created}\text{"}$ and postcondition $status'_a = \text{"}\texttt{offered}\text{"} \wedge P(\mathscr{A} \setminus \{status_a\})$;

2. A service $offered-to-started_a$ with precondition $status_a = \text{"}\texttt{offered}\text{"}$ and postcondition $status'_a = \text{"}\texttt{started}\text{"} \wedge P(\mathscr{A} \setminus \{status_a\})$;

3. A service $started-to-completed_a$ with precondition $status_a = \text{"}\texttt{started}\text{"}$ and postcondition $status'_a = \text{"}\texttt{completed}\text{"} \wedge P(\mathscr{A} \setminus (\{status_a\} \cup O_a)) \wedge \psi_a$, with ψ_a being the postcondition on the output attributes O_a;

4. A service $started-to-failed_a$ with precondition $status_a = \text{"}\texttt{started}\text{"}$ and postcondition $status'_a = \text{"}\texttt{failed}\text{"} \wedge P(\mathscr{A} \setminus \{status_a\})$;

The transitions from the state `failed` will be describe in the next subsection. Now we define how composite activities compute preconditions for their children activities. First, we define the concept of *completing condition*. Every activity a has a different completing condition γ_a defined in the following way:

1. If a is basic activity then $\gamma_a = status_a =$ "`completed`";
2. If $a = \{\sigma, exc\_hnd\}$ is a subflow, let l be the last activity in σ, then $\gamma_a = \gamma_l$;
3. If $a = \{B\}$ is an AND-Split, let L be the set of the last activities in the sequences in B, then $\gamma_a = \bigwedge_{l \in L} \gamma_l$;
4. If $a = \{b_{true}, b_{false}, \chi\}$ is a XOR-Split, let l_{true} (resp. l_{false}) be the last activity of b_{true} (resp. b_{false}), then $\gamma_a = l_{true} \vee l_{false}$.

Now we can define the preconditions of all basic activities recursively on the hierarchy of composite activities. Clearly, every composite activity has a precondition too.

1. In a subflow $a = \{\sigma, exc\_hnd\}$, $\pi_{\sigma_0} = \pi_a$, then $\forall i > 0(\pi_{\sigma_i} = \gamma_{\sigma_{i-1}})$;
2. In an AND-Split $a = \{B\}$, for every branch $\langle b_i \rangle \in B$, we have $\pi_{b_0} = \pi_a$ and $\forall i > 0(\pi_{b_i} = \gamma_{b_{i-1}})$;
3. In a XOR-Split $a = \{b_{true}, b_{false}, \chi\}$, let $\langle t_i \rangle = b_{true}$ and $\langle f_i \rangle = b_{false}$, then $\pi_{t_0} = \pi_a \wedge \chi \wedge status_{f_0} =$ "`created`" and $\forall i > 0(\pi_{t_i} = \gamma_{t_{i-1}})$. Also, $\pi_{f_0} = \pi_a \wedge \neg\chi \wedge status_{t_0} =$ "`created`" and $\forall i > 0(\pi_{f_i} = \gamma_{f_{i-1}})$.

The root sub-workflow r in the hierarchy has a precondition $\pi_r = true$ and an extra service is added at the end that updates the global status.

- A service *global_complete* with precondition γ_r and postcondition $global\_status' =$ "`completed`" $\wedge P(\mathscr{A} \setminus \{global\_status\})$;

Note. The introduction of activities that always fail or that never fail is trivial.

5.2.3 Exception handling services

The services that we defined for basic activities will never change the state of a basic activity that failed. We have now to account for th exception handling policies introduced in Subsection 5.1.4.

We first deal with the case of manual basic activities with the `reallocate` policy. To handle these activities we simply add a service:

- A service $failed-to-created_a$ with precondition $status_a =$ "`failed`" and postcondition $status'_a =$ "`created`" $\wedge P(\mathscr{A} \setminus \{status_a\})$;

Whenever a basic activity fails its exception gets forwarded to its parent subflow and handled with the parent policy. This is handled together with the restart case in the following way. First we define the concept of scope. We call the *scope* S_a of an activity a the following set of basic activities:

- If a is a basic activity then $S_a = \{a\}$;
- If $a = \{\sigma, exc\_hnd\}$ is a subflow, then $S_a = \bigcup_{s\in\sigma} S_s$;
- If $a = \{B\}$ is an AND-Split, then $S_a = \bigcup_{s\in B} S_s$;
- If $a = \{b_{true}, b_{false}, \chi\}$ is a XOR-Split, then $S_a = \bigcup_{s\in b_{true}} S_s \cup \bigcup_{s\in b_{false}} S_s$.

Whenever a basic activity a fails, we call anc_a the closest ancestor (possibly a itself) s.t. the exception handling policy of anc_a is `restart`. If there is no such ancestor we add a service that fails the whole workflow:

- A service $failed-to-globalfail_a$ with precondition $status_a =$ "`failed`" and postcondition $global\_status' =$ "`failed`" $\wedge P(\mathscr{A} \setminus \{global\_status\})$

If anc_a exists, intuitively, if it is a basic activity, we just restart a, if anc_a is a subflow we restart the whole subflow. We implement this strategy by introducing the following service for every basic activity with a exception handling policy of `fail` or `restart`.

- A service $failed-to-restart_a$ with precondition $status_a =$ "`failed`" and the following postcondition:

$$\bigwedge_{a \in S_{anc_a}} (status'_a = \text{"}\texttt{created}\text{"} \wedge \bigwedge_{d \in O_a} d = null) \wedge P(\mathscr{A} \setminus \bigcup_{a \in S_{anc_a}} (\{status_a\} \cup O_a)$$

Note how restarting a scope, not only changes the state of all the basic activities in the scope of the restarted subflow, but also resets the output attributes of those activities. We will see in Section 5.3 how this is crucial to ensure feedback-freedom of the business processes expressed as AWEs.

5.2.4 Semantics

In the previous subsection we defined, starting from an AWE business process, a set of attributes $\mathscr{A}$ and a set of services for each basic activity b , that we will call $\mathscr{S}_b$, that includes the transitions from the `created` state to `completed` or `failed` state, along with the service implementing the exception handling policies.

Since our service-based model assumes infinite runs we add two dummy services $\{completed\_dummy, failed\_dummy\}$ that propagates the final state of the workflow (completed or failed):

- A service *completed_dummy* with precondition $global\_status =$ "`completed`" and postcondition $P(\mathscr{A})$;
- A service $failed\_dummy$ with precondition $global\_status =$ "`failed`" and postcondition $P(\mathscr{A})$;

Definition 5.2.1. *Let $\mathscr{DB}$ be the database used in an AWE business process BP with basic activities in $\mathscr{B}$. Then we define an artifact schema S to be $\{\mathscr{A}, \mathscr{DB}\}$. We have then that the* service based translation *of BP is an artifact system $\{S, \{\mathscr{S}_b | b \in \mathscr{B}\} \cup \{completed\_dummy, failed\_dummy\}, \Pi\}$, with:*

- *S being the artifact schema $\{\mathscr{A}, \mathscr{DB}\}$;*
- *$\{\mathscr{S}_b | b \in \mathscr{B}\}$ being all the services defined for the basic activities in BP; and*

- Π *be the following precondition:*

$$global\_status = \text{"}\texttt{created}\text{"} \wedge \bigwedge_{a \in \mathscr{B}} (status_a = \text{"}\texttt{created}\text{"} \wedge \bigwedge_{o \in O_a} o = null)$$

The runs of *BP* are then all and only the runs of its service-based translation.

5.3 Expressive power comparison

In this section we explore the expressive power of AWEs. Let us first modify the AWE model by not resetting all output attributes whenever a subflow is restarted. We call this model AWE*. It turns out that AWE* is as powerful as the sevice-based model described in Chapter 2, using only a constant number of extra attributes. More formally, we say that two runs ρ and ρ' are equal up to *constant-stutter* iff there exists a linear function $f(i)$ s.t. $\forall i(\rho(i) = \rho'(f(i)))$. Then we have:

Theorem 5.3.1. *Artifact systems have the same expressive power of AWE*, up to constant-stutter.*

Proof: First, we note that AWE* semantics is expressed as a service-based artifact system, so we just need to prove that any service-based business process p can be expressed as an AWE*. The first observation is that we can rewrite p in p' that uses a single service. Let Σ be the set of services in p, we define $\Sigma' = \{s\}$ as the set of services in p', where s has the following precondition:

$$\pi_s = \bigvee_{t \in \Sigma} \pi_t.$$

In order to define the postcondition of s we define a set called $C(\Sigma)$. The formulas in $C(\Sigma)$ are implication where the left side is of the following form. Let $\sigma \in 2^{\Pi}$, with Π being the set of all preconditions of the services in Σ, and Ψ_σ be the set, given a $\sigma \in 2^{\Pi}$, of all postconditions ψ_i of a service i s.t. $\sigma(\pi_i) = 1$, then we the following postcondition for s:

$$\bigwedge_{\sigma \in 2^{\Pi}} \bigwedge_{\psi_i \in \Psi_\sigma} ((\bigwedge_{\sigma(\pi)=1} \pi \wedge \bigwedge_{\sigma \pi = 0} \pi) \rightarrow \psi_i)$$

It is easy to see that every run of p is a run of p' and vice versa.

Let us consider the case with no global precondition in p (the extension to this case is trivial). Now, we can build an AWE* business process in this way. The data attributes are all the artifact attributes $\bar{x}_p$ of p. Let r be the root worflow with exception handling policy `restart` and a sequence of activities that includes two activities a_p and a_f. The activity a_f is a basic activity that always fails and has an exception handling policy of `fail`. The activity a_p is a basic activity that never fails and has $\bar{x}_p$ as output attributes, $\pi_s \wedge \psi_s$ as postcondition, and an exception handling policy of `fail`. It is easy to see that, excluding the status attributes for the global state and the two basic activities, every run ρ_A of the above AWE* process has a corresponding run $\rho_{p'}$ of p' s.t. $\rho_A(i) = \rho_{p'}(5i+2)$ and vice versa. The stuttering accounts for the instants that the AWE* process uses to transition the states of the activities and restarting the root subflow. Note also that every blocking run of p' has a corresponding blocking run in the AWE* system and vice versa. This follows from the fact that a_p never fails, so in any instant when its state is `started` the only available service is its completion service that blocks the run as its postcondition is unsatisfiable. □

Now we describe how AWEs relate to the feedback freedom concept. First we introduce two classes of first-order conditions:

Definition 5.3.2. *We say that a first-order formula ϕ on the attributes of an AWE is* hierarchy-safe *iff for each pair of attributes a and b in ϕ, let b_a (resp. b_b) be the basic activity that has a (resp. b) as a status or output attribute, then b_a is in the scope of an ancestor of b_b or viceversa.*

And,

Definition 5.3.3. *We say that a first-order formula ϕ on the attributes of an AWE is* hierarchy-safe *w.r.t. a basic activity b iff for each attribute a in ϕ, let b_a be the basic activity that has a as a status or output attribute, then b is in the scope of an ancestor of b_a.*

Intuitively, hierarchy-safe conditions limit the interactions between siblings of the subflow hierarchy. We will show that forcing conditions in the specification and temporal properties to be hirarchy-safe guarantees feedback-freedom.

Definition 5.3.4. *We call AWE^s the AWE where:*

- *any postcondition of a basic activity a is hierarchy-safe w.r.t. a; and*
- *any XOR condition is hierarchy safe.*

We then have:

Theorem 5.3.5. *Let Γ be an AWE^s, and φ be a temporal property whose first-order components are hierarchy safe, then (γ, φ) is feedback-free.*

Proof: Let us consider, in a computational graph of an AWE^s two different equivalence classes $[x_i]$ and $[x_j]$ of the same artifact variable in the same connected component. Let i and j be the smallest instants in the respective spans. We know, also, that, since x has a different value, it means that an ancestor subflow has been restarted between i and j. Moreover, the existence of a path between $[x_i]$ and $[x_j]$ means that it has to go through an attribute y belonging to an ancestor that was not restarted. It follows that the span of $[y_i]$ has to include the spans of both $[x_i]$ and $[x_j]$. Let us assume by contradiction, that this is not the case. This would mean that y was reset in an instant included in the span of either $[x_i]$ or $[x_j]$. This is not possible as, being y an attribute of an ancestor, it would have reset the value of x. □

This Theorem gives a useful necessar condition for feedback freedom. We will exploit this property when randomly generating feedback free systems in Chapter 7.

Part II

Feasibility Study

6 Verifier Implementation

This chapters details the implementation of a verifier prototype based on the ideas presented in Chapter 4. We will describe the implemetation of the algorithm introduced in Section 4.3, that supports verification of artifact systems with database schemas with key constraints and arithemetic operations, incurring in the hyperexponential complexity only when arithmetic is involved. The architecture of the prototype is general enough to support the implementation of all the verification algorithms in Chapter 4. During the description of the design we will highlight how to modify the code to implement the other algorithms.

The chapter starts with a description of the high-level architecture of the prototype, before delving into the detailed explanation of the implementation of the verification algorithm. A final section introduces two optimizations that compensate for the inefficiencies of the first-cut implementation.

6.1 Architecture

In this section we will describe the high level architecture of the prototype verifier. We implemented the verifier prototype in Java making use of external libraries whenever possible. The prototype takes as input an artifact system as presented in Chapter 2 and an LTL-FO property property to verify. Its output is a boolean value containing the result of the verification, and, in case the property is not satisfied, it provides a symbolic run acting as a counterexample.

Besides some utilities (e.g. user-friendly interfaces), the prototype is divided into three main modules: parser, pre-verification setup, and verifier. These modules have different dependencies from each other and from external libraries.

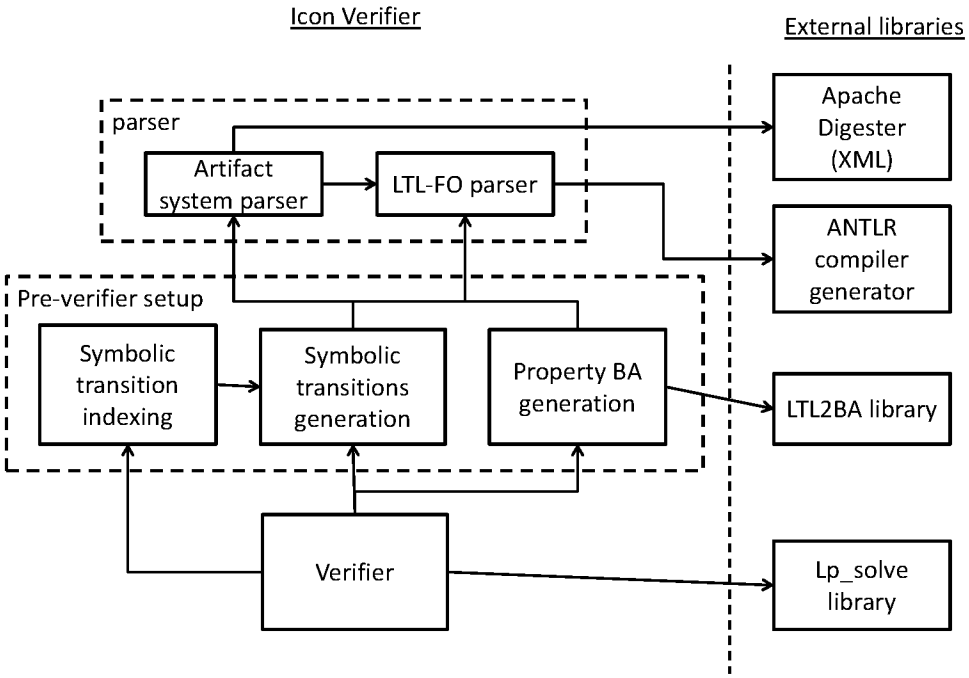

Figure 6.1: Architecture of the prototype implementation, with dependency relation

6.1.1 Parser

The parser module loads the description of an artifact system into the object structure used by the prototype. The prototype accepts business process specifications in the form of a single xml file that details the artifact attributes, database schema, global precondition, and list of services. The parser module uses the Apache Digester [dig] library to load the contents of the xml file into a memory object structure. In the xml specification there are first-order logical formulas that are parsed into object structure with the use of a custom parser built with ANTLR [ant]. This parser handles first order and LTL-FO formulas and performs a first layer of semantic checks and normalizations. The most important features of the parser/semantic checker are:

- support for numerical and string constants;
- support for database predicates;

- support for linear arithmetic constraints;
- support for LTL and logical operators;
- on-the-fly normalization of schemas with unary keys;
- semantic checks of database predicates in formulas (name and arity);
- conversion of formulas to the normalized database schema;
- for first-order formulas, rewriting to DNF;
- for LTL formulas, identification of first-order components;

The output of the artifact system parser submodule is an object structure that represents a system where every service has preconditions and postconditions expressed in DNF and that refer to the correct set of predicates and arithmetic constraints. The output of the LTL-FO parser, when it parses an temporal property, is an object structure representing an LTL-FO formula with all the first-order components expressed in DNF and that refer to the correct set of predicates and arithmetic constraints.

6.1.2 Pre-verification setup

The pre-verification module computes the information needed by the verification algorithm, performs some computation that can be done in advance, and sets up additional data structures.

The information needed by the verifier are:

- the set of symbolic transitions; and
- the Büchi automaton of the negation of the property.

The set of symbolic transitions is computed using the procedure described in Section 2.3. Unsatisfiable symbolic transitions are discarded; and for the remaining ones we precompute which first-order components of $\neg\varphi$ are implied. This last information is useful as we do not have to compute this satisfiability in the internal loop of the verifier module.

As usual with verification algorithm based on model checking, the verifier module looks for a counterexample of φ, which means looking for a symbolic run satisfying $\neg\varphi$. The satisfaction of $\neg\varphi$ of a symbolic run is checked with the use of a Büchi automaton. The property BA generation submodule rewrites $\neg\varphi$ into $\neg\varphi'$, which substitutes first-order components with propositional variables. It then invokes the external library LTL2BA [jpf] from the Java Pathfinder project to generate the $BA_{\neg\varphi}$, used by the verifier module.

An additional index data structure is built in order to speed up access to symbolic transitions in the verifier internal loop. The details of this index are specified in Subsection 6.3.2.

6.1.3 Verifier

The verifier module implements the verification algorithm as described in Section 6.2. The implementation uses the external library lp_solve [lps] to verify the satisfiability of systems of linear inequalities. Lp_solve is implemented in C++ and we use the provided JNI interface to access it from Java. In the module implementation, a simple Java class acts as interface performing data conversion from the Java object structure representing the linear inequalities atoms to the vector representation required by the JNI interface of lp_solve.

Special care had to be taken in the translation of strict inequalities of the form $>$, $<$ and $\neq$. Lp_solve does not support those inequalities. This is because the 128bit double datatype used in the C++ implementation does not have unlimited precision. This resulted in the following rewritings:

- $l(\bar{x}) > c$ becomes $l(\bar{x}) \geq c + g$;
- $l(\bar{x}) < c$ becomes $l(\bar{x}) \leq c - g$;

with $l(\bar{x})$ being a linear combination for variables in $\bar{x}$ and g the chosen granularity (2^{-16382} in our implementation). We will see in Subsection 6.2.2 how we avoid calling lp_solve with $\neq$-inequalities.

6.2 Verification algorithm

The theoretical foundations of the verification algorithm implemented are described in Section 4.3. At a high-level, there is the idea, common to all model checking techniques, to look for a counterexample, i.e. a run that satisfies the negation of the property to verify. Since we work with infinite state systems, we have to work with symbolic representations of runs. We then exloit the result in Lemma 4.3.8, that we restate for convenience:

Lemma 6.2.1. *In an artihmetic-key-feedback-free artifact system and property with key constraints and arithmetic, there exists a run ρ satisfying $\neg\varphi$ iff*

1. *there exists a symbolic run prefix $\langle\psi_i\rangle_{i<j+n}$ s.t. $[\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n}) \wedge \bar{x}_j = \bar{x}_{j+n}$ is satisfiable, and*

2. *for such $\langle\psi_i\rangle_{i<j+n}$, there exists a run $\{q_i\}_{i\leq j+n}$ of $B_{\neg\varphi}$ on $\{\sigma_i\}_{i\leq j+n}$ (i.e. the run of truth assignments for FO components in $\neg\varphi$ given the run $\{\psi_i\}_{i<j+n}$) such that for some accepting state r. $q_j = q_{j+n} = r$.*

Our algorithm searches the space of symbolic runs (i.e. sequences of symbolic transitions $\langle\psi_i\rangle$) looking for a run satisfying the above conditions. In order to perform the search, we maitain a state composed of a reduced form of the inherited constraints up to a certain instant and the current state in the Büchi automaton of $\neg\varphi$.

The algorithm structure is very similar to the nested depth first search used in standard model checking. The standard depth first search algorithm explores all run prefixes of a transition system (the artifact system in our case) while running the BA of the negated property on the generated prefix. Then, every time the BA is on a final state, it spawns another depth first search that looks for a cycle back to the originating state.

The differences of our algorithm w.r.t. the standard depth first search pertain to the maitenance of the reduced inherited constraints and the acceptance conditions.

- The procedure used to compute $red^m([\eta]|_i)$, given a current reduced form $red^m([\eta]|_{i-1})$ and a symbolic transition ψ_i, is the one detailed in Subsection 4.3.1;

- Satisfiability is checked by separating the arithmetic portion of the inherited constraints from the database part. The details are in Subsection 6.2.2;

- When looking for a cycle at instant j with a BA final state s_j, the algorithm starts maintaining inherited constraints that aways keep $[\bar{x}_j]$ as free variables in addition to the current ones. Then, when in the nested search, it just looks for the first configuration s.t. the state is s_j and $red^m([\eta]|_j^{j+n}(\langle\psi_i\rangle_{i\leq j+n})) \wedge [\bar{x}]_j = [\bar{x}]_{j+n}$ is satisfiable.

Note how the final condition comes from Lemma 4.3.8. A more detailed description of the verification algorithm, along with the pseudocode, is given in Subsection 6.2.1. Clearly, this is an implementation of the algorithm described in Subsection 4.3. We also implemented the algorithms in Section 4.1 (for artifact systems with no dependencies and no arithemtic) and 4.2 (for artifact systems with unary keys and no arithmetic). The difference between the implementation of these algorithm and the one presented here is simply in the procedure for computing the reduced form (as explained in Chapter 4) and checking satisfiability (eliminating the arithmetic solver and no handling of key constraints).

6.2.1 Algorithm pseudocode

Algorithms 1 and 2 show the pseudocode of our algorithm. In this subsection we explain the pseudocode in detail.

The main procedure, lines 1-8, initializes the data structures for the search and calls the `dfs` procedure starting from every possible dsjunct of the global precondition. In our algorihm the inherited constraints (referred to as *icon* in the pseudocode) are always with equivalence class variables (i.e. we store the reduced form of $[\eta]$ and not of η). We maitain a simple data structure keeping track of the correspondence between the current artifact variables at instant i, $\bar{x}_i$ and the equivalence classes $[\bar{x}]_i$ in the *icon* representation. For clarity's sake, the maintenance of this data structure is omitted in the pseudcode. Note that line 5, calls the procedure `create_eqclasses`, to put the global precodition disjunct in the above defined form. Line 7 invokes then the depth first search algorithm starting from the state containing the precondition disjunct and the initial state of $BA_{\neg\varphi}$.

The depth first search procedure generates all possible satisfiable prefixes and, upon reaching a final state for $BA_{\neg\varphi}$, it spawns a nested depth-first search for a cycle as defined at the beginning of the section. The input of the `dfs` procedure is a state $\langle icon(\bar{[x]}), ba\_state \rangle$. Lines 12-13, makes sure that we do not visit a state more than once. Note that we consider $\langle icon(\bar{[x]}), ba\_state \rangle \in$ visited if there is a state $\langle icon'(\bar{[x]}), ba\_state \rangle$ where *icon* and $icon'$ are syntactically equal up to a one-to-one renaming of existentially quantified variables. If *ba_state* is a final state, lines 14-17, spawn a search for a cycle by calling the procedure `cycle`, described below. Lines 18-25 generate all possible next states $\langle icon_n, s_n \rangle$. In order to do this, line 18 iterates over all the possible symbolic transitions, while lines 19-20 use the procedure detailed in Subsection 4.3.1 to create the reduced form of the inherited constraints of the concatenation of the current prefix and ψ. Line 21 prunes unsatisfiable prefixes. Lines 22-25 iterates over all possible transitions from *ba_state* in $BA_{\neg\varphi}$. Line 23 prunes transitions that are not satisfied by the current symbolic transition ψ, and line 24 continues the search.

The `cycle` procedure is shown in Algorithm 2. It takes as input a state, a final state from which the cycle starts (called *knot*) and the set of variables $\bar{[x_j]}$ that were current when the cycle started. Analogously to `dfs` the `cycle` procedure continues generating satisfiable prefixes. There are two differences with `dfs`, though. The first is that, when computing the reduced form in line 7, it considers $\bar{[x']} \cup \bar{[x_j]}$ as free variables. This is necessary to maitain the reduce form of the cycle inherited constraints as defined in Chapter 4. The other difference is then, before recursing in line 11, line 10 checks if the next configuration creates a cycle in $BA_{\neg\varphi}$ by reaching back to the *knot* state and if $redicon(\bar{[x]}) \wedge \bar{[x]} = \bar{[x_j]}$ is satisfiable. Note how this directly corresponds to condition 1 of Lemma 4.3.8.

6.2.2 Checking satisfiability

In Lemma 3.2.3 we proved that we can check satisfiability independently of the database an arithmetic portions of the inherited constraints. The technique outlined in Lemma 3.2.3, however, assumes to check satisfiability for all the possible equality types of the set of common variables. Our implementation exploits the fact that we only allow key dependencies in the following way. As in Section 4.3, let us call $[\eta_{db}]$ the subset

of terms that either refer to database predicates or to equalities and $\neq$-inequalities, and $[\eta_a]$ the subset with only equalities, $\neq$-inequalities and arithmetic terms. The main idea is to identify the only variables whose equality type can affect the satisfiability of $[\eta_{db}]$. Since we only have key constraints, an equality $x = y$ can cause a contradiction in $[\eta_{db}]$ only if $[\eta_{db}]$ contains:

1. $x = c_1$ and $y = c_2$, with c_1 and c_2 different constants; or
2. $p(\bar{x})$ and $\neg p(\bar{y})$, with $x \in \bar{x}$ at position i and $y \in \bar{y}$ at position i; or
3. $x \neq y$; or
4. $p(x,w)$ and $p(y,z)$, with the first attribute being a key for predicate p.

Analogously, an inequality $x \neq y$ can cause a contradiction in $[\eta_{db}]$ only if $[\eta_{db}]$ contains:

1. $x = y$; or
2. $x = c$ and $y = c$; or
3. $p(w,x)$ and $p(z,y)$, with the first attribute being a key for predicate p.

We call *critical pairs* the set of pairs of variables that satisfy any of the above condition.

It follows that we only need to check only the equality types for the critical pairs in order to check satisfiability of $[\eta]$.

Lemma 6.2.2. *Let C be the set of critical pairs, then* $[\eta]$ *is satisfiable iff there exist an equality type* $eq(C)$ *for C s.t.* $[\eta_{db}] \wedge eq(C)$ *is satisfiable and* $[\eta_a] \wedge eq(C)$ *is satisfiable.*

Our algorithm identifies the critical pairs, it then generates for all possible equality types and checks independently the satisfiability of $[\eta_{db}] \wedge eq(C)$ and $[\eta_a] \wedge eq(C)$. Remember that we check the satisfiability of $[\eta_a] \wedge eq(C)$ using the lp_solve library (cfr. 6.1.3), which does not support $\neq$ constraints. It follows that we actually generate equality types in the form of constraints of $=$, $>$ and $<$, and everytime we check satisfiability of the database portion we simply consider $>$-constraints and $<$-constrains as $\neq$-inequalities.

In order to check the satisfiability of $[\eta_a] \wedge eq(C)$, we simply call the external solver. The following details the satisfiability check of $[\eta_{db}] \wedge eq(C)$.

Checking satisfiability of $[\eta_{db}] \wedge eq(C)$

Let us assume first that we have no data dependencies. Our algorithm computes the equivalence classes eq of $[\eta_{db}] \wedge eq(C)$ w.r.t. to equality predicates. If x is a variable or a constant in $[\eta_{db}] \wedge eq(C)$, we call $[x]$ its equivalence class. At this point, if there is a contradiction of the form $[x] = c_1 \neq c_2$ it is easily identified, by looking for equivalence classes with more than one constant.

After this first step we scan $[\eta_{db}] \wedge eq(C)$ looking for contradictions of the form:

1. $x \neq y$, with $[x] = [y]$; or
2. $p(\bar{x}) \wedge \neg p(\bar{y})$, s.t. $[\bar{x}]|_k = [\bar{y}]|_k$ with $[\bar{x}]|_k$ (resp. $[\bar{y}]|_k$) being the vector of equivalence classes of the variables in the key attributes of $\bar{x}$ (resp. $\bar{y}$).

When we introduce key constraints, we may have some additional equalities that are introduced by the data dependencies. In order to take those into account, we modify the previous procedure simply by considering the additional equalities in the computation of the equivalence classes. We call *application* of a key constraint on predicate p w.r.t. equivalence classes eq, the generation of the equality $w = z$ from the literals $p(x,w) \wedge p(y,z)$ with $[x] = [y]$ w.r.t. eq. Our algorithm then proceeds in the following way:

1. computes the equivalence classes eq of $[\eta_{db}] \wedge eq(C)$ w.r.t. to equality predicates;
2. computes all the equalities implied by the key constraints by looking for applications of key constraints on $[\eta_{db}] \wedge eq(C)$ until no more equalities can be added to eq;
3. if eq has a contradiction, output $false$;
4. looks for a contradiction in $[\eta_{db}] \wedge eq(C)$ w.r.t. eq as described above: if found, output $false$, otherwse ouput $true$.

Theorem 6.2.3. *The above procedure returns $true$ iff $[\eta_{db}] \wedge eq(C)$ is satisfiable.*

Proof: The procedure is simply a variation of the chase. Instead of materializing different tuples as a result of the application of a key constraint, we keep track of the equalities

in the equivalence classes. Then we verify that we did not generate any contradiction by checking the atoms w.r.t. the equivalence classes. □

6.3 Optimizations

In this section we describe two optimizations that help speed up the execution of the verification algorithm by optimizing the two most common operations performed in the inner loop of our nested depth first search algorithm. The first one is used to enable the use of hashing when checking if a state has been visited. The second exploits a heuristic to greatly prune the set of symbolic transitions to be tried when looking for a satisfiable prefix.

6.3.1 Inherited constraint hashing

In our verification algorithm we have to check at every iteration if a state has been already visited (line 12 of Algorithm 1). This check involves computing if two representations of the inherited constraints are equivalent. The notion of equivalence is complicated by two facts: 1) the inherited constraint representations use equivalence classes as variables, and 2) two inherited constraints are considered equivalent iff there exists a one-to-one renaming of the existentially quantified variables that makes the two formulas syntactically equal up to literal reordering.

The use of equivalence classes implies that a free variable $[x_j]$ might represent more than a single original free variable, e.g. it might refer to variables x_j and y_j. It follows that when comparing two inherited constraints, we consider two free variables to be equal if their equivalence class is associated to the same set of artifact attributes value in the current instant. We say that $[\eta_1] \simeq [\eta_2]$ whenever two inherited constraints are equal up to literal reordering and considering free variables equal as described above.

It follows that, the algorithm that checks equivalence ($\approx$) of two inherited constraint representations $[\eta_1]$ and $[\eta_2]$ performs the following operations:

1. computes the set M of one-to-one renamings of the existentially quantified variables of $[\eta_1]$ into existential variables of $[\eta_2]$;

2. if $\exists \mu \in M$ s.t. $\mu([\eta_1]) \simeq [\eta_2]$, then $[\eta_1] \approx [\eta_2]$

In order to avoid the expensive linear scan of the whole set of visited states at every iteration, we designed a hashing function compatible with this notion of $\approx$-equivalence. Assuming the existence of hashing functions for sets and variables (we used the standard Java implementation for hashes of sets and strings, respectively), we define the following hashing functions $h()$, given the current instant j:

- with $[x]$ a free variable, $hash([x]) = hash(set_a([x]))$, with $set_a([x])$ being the subset of $[x]$ of variables of instant j;
- with $[y]$ an existential variable, $hash([y]) = 0$;
- $hash(p([\bar{x}]))$, normal implementation of an hashing function for literal using the above hashes for the variables;
- $hash([\eta]) = hash(set_l([\eta]))$, with $set_l([\eta]))$ being the set of literals in $[\eta]$.

Storing the visited states in a hash table resulted in a huge improvement in the performance of our algorithm.

6.3.2 Symbolic transition index

In our verification algorithm we prune unsatisfiable symbolic prefixes in order to limit the search space, e.g. line 5 and 21 of Algorithm 1. As described in subsection 6.2.2, the satisfiability check can be very complex, and sometimes it is possible to quickly exclude symbolic transitions whose concatenation with the current inherited constraints will surely result in a contradiction.

Our technique considers the contradictions that come from equalities and $\neq$-inequalities with constants. For instance, if an inherited constraint includes the literal $[x_i] = c$, with $[x_i]$ being a free equivalence class variable that contains the current value of attribute x, then no symbolic transition containing $x \neq c$ (or $x = c'$, with $c' \neq c$) can be concatenated. By extending this reasoning, we will show how to build an index structure for symbolic transitions that, given a set of equalities and $\neq$-inequalities with between current attribute values and constants, retrieves a superset of the symbolic transitions that will be satisfiable together with the current inherited constraints.

Let us call C the set of all the constants in the symbolic transitions, the index structure contains, for each formula $f : x = c$ (or $x \neq c$), with $x \in \bar{x}$ and $c \in C$, the list of all symbolic transitions ψ s.t. $\psi \wedge f$ is satisfiable. It is clear that the size of the index is polynomial w.r.t. the number of symbolic transitions, and, knowing C, it can be built with a single scan of the symbolic transition set.

At a high level, in order to prune the *candidate symbolic transitions* (i.e. set of symbolic transitions on which we apply the full satisfiability algorithm), we exploit the above structure by taking the intersection of the sets associated with all the equalities and $\neq$-inequalities in the current inherited constraints. More formally, let $[\eta]$ be the current inherited constraints, and let $e : x = / \neq c$ be an equality and inequality with $x \in \bar{x}$ and $c \in C$, we say that $[\eta]$ implies e iff $[x] = / \neq c$ is an atom in $[\eta]$. Now, let S be the set of all symbolic transitions, let $index(e)$ be the set of symbolic transitions returned by the index structure for equality (or $\neq -inequality$) e, the set of candidates for $[\eta]$:

$$S \cap \bigcap_{e \in \{e \text{ implied by } [\eta]\}} index(e).$$

Algorithm 1: Verifies that $\Gamma \models \varphi$, part 1.

Input: $\Psi_{(\Gamma,\varphi)}$ is the set of symbolic transitions of (Γ, φ),
Π_Γ is the set of disjuncts of the DNF of the global precondition of Γ,
$BA_{\neg\varphi} = \{Q, i, \delta, F\}$ is the BA of $\neg\varphi$

```
1  begin
2      visited = ∅
3      found = false
4      foreach π ∈ Π_Γ do
5          icon([x̄]) ⇐ create_eqclasses(π)
6          if satisfiable(icon([x̄])) then
7              dfs(⟨icon([x̄]), i⟩)
8      return ¬found
9  end
10 procedure dfs(⟨icon([x̄]), ba_state⟩)
11 begin
12     if ⟨icon([x̄]), ba_state⟩ ∈ visited then return
13     visited ⇐ visited ∪{⟨icon([x̄]), ba_state⟩}
14     if ba_state ∈ F then
15         visited2 ⇐ ∅
16         cycle(⟨icon([x̄]), ba_state⟩, ba_state, [x̄])
17         if found then return
18     foreach ψ ∈ Ψ_Γ do
19         newicon([x̄], [x̄′]) ⇐ icon([x̄]) ∧ create_eqclasses(ψ)
20         redicon([x̄]) ⇐ reduce(newicon([x̄], [x̄′]), [x̄′])
21         if not satisfiable(redicon([x̄])) then continue
22         foreach ⟨ba_state, θ, next⟩ ∈ δ do
23             if ψ ⊭ θ then continue
24             dfs(⟨redicon([x̄]), next⟩)
25             if found then return
26 end
```

Algorithm 2: Verifies that $\Gamma \models \varphi$, part 2

Input: $\langle icon([\bar{x}]), ba\_state \rangle$ is current state(inherited constraints and BA state),

knot is the final BA state that we are trying to reach,

$[\bar{x_j}]$ are the free variables in the first instant of the cycle.

```
1  procedure cycle(⟨icon([x̄]), ba_state⟩, knot, [x̄_j])
2  begin
3      if ⟨icon([x̄_j], [x̄]), ba_state⟩ ∈ visited2 then return
4      visited2 ⇐ visited2 ∪ {⟨icon([x̄_j], [x̄]), ba_state⟩}
5      foreach ψ ∈ Ψ_Γ do
6          newicon([x̄], [x̄′]) ⇐ icon([x̄_j], [x̄]) ∧ create_eqclasses(ψ)
7          redicon([x̄]) ⇐ reduce(newicon([x̄], [x̄′]), [x̄′] ∪ [x̄_j])
8          if not satisfiable(redicon([x̄])) then continue
9          foreach ⟨ba_state, θ, next⟩ ∈ δ do
10             if ψ ⊭ θ then continue
11             if next = knot and satisfiable(redicon([x̄]) ∧ [x̄] = [x̄_j])
               then found ⇐ true
12             cycle(⟨redicon([x̄]), next⟩, knot, [x̄_j])
13             if found then return
14 end
```

7 Experimental evaluation

In this chapter we describe the experimental evaluation we performed on the verifier prototype we described in Chapter 6. Since we were not able to find a large collection of real-world data-aware business process specifications, we resorted to automatic generation. In order to generate realistic specifications and properties, we devised a generation algorithm based on the high-level model described in Chapter 5 and statistics extracted from real world business process specifications. The first part of this chapter describes our data generation procedure. Then, after introducing the execution enviroment, we describe the experiments we performed and the results we obtained in terms of running time of the verification of various properties.

7.1 Business process generation

The generation of busines processes follows the high-level model developed in Chapter 5 and then exploits statistics extracted from [TLR07] in order to generate realistic processes. The generation algorithm has two phases. The first follows the hierarchical organizations of acyclic workflow with exceptions (AWE) to generate the structure of the workflow. The second phase generates the data conditions on XOR-Splits and basic activities that modify data attributes.

In order to generate the workflow structure we extracted from [TLR07] the frequencies of the following basic patterns: XOR-Splits (15%), AND-Splits (15%), and basic activities (70%). Given the number of basic activities to generate as input, we start generating patterns with the above mentioned frequencies. When we generate a composite activity (XOR-Splits, AND-Splits), the number of basic activities it contains is randomly picked from a uniform distribution of the number of activities we still have

to generate. A basic activity can be either manual or automatic. In our experiments we used a 50% chance as we could not extract this frequency from [TLR07]. Also, the patterns in [TLR07] do not contain the concept of sub-workflow, so at each generation step there is a chance (in our experiments we used 30%) to generate a sub-workflow, that is then treated like another composite activity.

Every activity is randomly chosen as being one that can fail or one that is assumed always to complete. This models the fact that in real world specifications, some exceptions are handled at a lower compared to business process specifications (e.g. most technical failures). In our experiments we assumed a 50% chance of handling exceptions at the business level. An exception handling policy is chosen for all subflows and all basic activities whose failure has to be handled at the business process level. The policy is chosen randomly between: restart, reallocate (for manual basic activities) and fail. Once all this is generated we have a specification of the workflow structure.

The second phase of the generation adds the data conditions to the generated workflow structure. The conditions have to be generated for XOR-Splits branches, and for basic activities that modify data attributes. The difference between these conditions is that the conditions in basic activities have to mention the output attribute of activity. In order to generate feedback-free systems, both kinds of conditions have to be hirarchy safe (Definitions 5.3.2 and 5.3.3). We assumed a single modified data attribute for each basic activity. The database schema is randomly generated in an uniform way, generating either binary tables with unary keys or tables with no keys. In our experiments we generated different databases with 6 tables and maximum arity of 5. The logical conditions are generated from a uniform distribution of logical connectors (i.e. $\wedge$, $\vee$, $\neg$, $\rightarrow$) and a uniform distribution of atoms of the following kinds:

- equality atoms: $att_1 = att_2$ or $att = constant$, with att, att_1 and att_2, either data or status attributes;
- database atoms: $table(\bar{a}$, with $\bar{a}$ a vector of data attributes and constant;
- arithmetic atoms: $\bar{c}^T \cdot \bar{a} = t$, with $\bar{c}$ a vector of coefficients, $\bar{a}$ a vector of data attributes and t a real number.

In our experiments we limited the attributes in arithmetic operations to 4.

Example 7.1.1 In Figure 7.1 we show an example of an AWE generated by our algorithm. The boxes represent basic activities. When they contain a condition it means that they perform a data operation.

The dashed edges represent exception handling transitions. Note how the basic activities with an exception handling policy of "fail", are all connected to the first activity of their subflow with a dashed edge labeled "restartParent". The condition on the XOR-Split is reported on the ourward edges of the "XOR-Split" node. Note, also, the nesting of composite activities like AND-Splits and XOR-Splits.

7.1.1 Complexity

In order to characterize the complexity of the business process we generate, we made use of two metrics. The first [McC76] is the standard Cyclomatic Complexity, also known as McCabe index. It measures the complexity of the flow of a program and it is directly related to complexity of testing a paticular piece of software. Intuitively, it is related to the number of execution paths that can arise in a certain procedure. The reason for choosing this metric comes from the fact that it is the standard in software engineering and there are widely accepted standard to what constitutes good programming practices for testability and understandability.

A problem with cyclomatic complexity is that is was developed for single-threaded software. Workflows, on the other hand, support parallelism, which is completely unaccounted for in the cyclomatic complexity. In order to solve this problem we measure the complexity of the business also with the Control Flow Complexity (CFC) introduced by Cardoso [Car05] specifically for business processes. This complexity is an extension of the cyclomatic complexity to workflows, and is widely accepted as a meaningful and validated metric for business processes [GL06].

While computing these metrics on standard workflows is straightforward, we have to account for the fact that our specifications included data operations. In order to include data operations we considered them as a a special case of splits. Borrowing from the possible paths interpretation of the cyclomatic complexity, we consider a declarative data operation with multiple disjuncts analogous to a if statement. This accounts for the fact that a tester should check individually the execution paths that result from each of

the disjuncts (and possibly their combinations).

In order to be conservative we computed the cyclomatic complexity of our specifications completely ignoring the parallelism, i.e. we assume all activities in AND-split to be executed sequentially. Then, we consider each XOR-Split and each basic activity whose failure has to be handled by the business process as if-statements. Data operations are considered as a case statement with a case for every combination of disjuncts. This is analogous to the accounting of OR-Splits in the CFC metric.

The CFC complexity is computed ina similar way except that each AND-Split increase the complexity by one. Also, every XOR-Split and basic activity with exception handling count as a case statement. Lastly, data operations are considered as OR-Splits, which is the same way as we account for them in the cyclomatic complexity.

7.1.2 Statistics

For our experiments we generated 50 business processes. In Figure 7.2 we report the distribution of the business processes w.r.t. cyclomatic complexity. Note that usually software quality guidelines force modules to have cyclomatic complexity of less than 10 [McC76], and consider any module with cyclomatic complexity above 50 to be 'untestable'.

In Figure 7.3 we report the distribution of the generated business processes w.r.t. control flow complexity. We will analyze the performance of our verification algorithm w.r.t. both these complexities.

7.2 Temporal properties generation

In order to generate temporal properties we followed the patterns and frequencies identified in [DAC98]. The work in [DAC98] surveys over 500 real-life specifications of temporal properties to be formally verified on finite state systems (theoretically equivalent to our contracts) and extracts recurrently appearing patterns that cover over 92% of the surveyed cases, along with their occurring frequencies. Ler φ and ψ be FO conditions, the patterns we use are:

Absence A condition φ is never true (a.k.a. safety property);

Existence A condition φ has to be true at some time (a.k.a. reachability property);

Universality A condition φ is always true (a.k.a. safety property);

Precedence An instant where φ is true must always be preceded by an instant where ψ is true;

Response An instant where φ is true must always be followed by an instant where ψ is true.

The work [DAC98] describes many variations of the above behaviors. In our experiments we generated only the most frequent ones, described above. The conditions used in the properties are generated in the same way as the one for the business processes.

Example 7.2.1 Some properties generated by our generator:

- **G** ((**NOT**(status_bi_man_per_164="`completed`")) $\rightarrow$
  F (**NOT** (status_bi_man_per_155="`offered`")));
- **G** ((-att_bi_aut_inf_132 + att_bi_aut_inf_131 + att_bi_man_per_126 = 1) $\rightarrow$
  F table0 (att_bi_man_per_136, att_bi_man_per_126));

The first is an instance of the response pattern for the business process in Figure 7.1. The second is another response pattern with conditions using database and arithmetic predicates.

7.3 Execution enviroment

We ran our experiments on a desktop PC with a AMD Phenom 2.2 Ghz quad-core CPU and 4Gb of RAM. We used Oracle JDK 1.6.0_21 on a Ubuntu 10.10 64bits installation.

7.4 Experiments

We measured the running times of three verification algorithms on a series of business process specifications and properties generated as specified in Sections 7.1 and 7.2. The algorithms we consider are the ones based on the theory of Chapter 4:

nodep for artifact systems with no keys and no arithmetic,

singlekey for feedback-free artifact systems with unary keys and no arithmetic, and

mixed for feedback-free artifact systems with unary keys and arithmetic.

Clearly, the results of the algorithms nodep and singlekey on inputs using keys and arithmetic include false negatives. However, they provide a good way to gauge the impact of the extra expressiveness on the performance of the verifier.

7.4.1 Scaling w.r.t. business process complexity

We now show the running times of the various verification algorithms w.r.t. the complexity of the business process. We ran the theree different algorithms on 20 temporal properties of 50 randomly generated feedback-free AWEs. In Figure 7.4 we show how the average running times of the various algorithms scale w.r.t. the Control Flow Complexity. Note how the average running time go from the seconds range for relatively simple specifications (up to CFC 60), and they reach the range of minutes for more complex ones (up to CFC 90).

In Figure 7.5 we show how the average running times of the various algorithms scale w.r.t. the Cyclomatic Complexity. Note how the Cyclomatic Complexity does not accurately reflects the complexity of the business process. Indeed, remember that Cyclomatic Complexity does not take paralellism into account. Moreover, we note that one of the three business processes with CC between 70-79 had nearly all conditions referring to arithmetic constrants. This explain th spike in the average running time for the mixed algorithm.

7.4.2 Discussion

From the results just presented, we argue that our approach is feasible for a useful range of business process specifications. Remember that cyclomatic complexity greater than 50 is considered ‘untestable’ [McC76], and our algorithm handles those complexities in the minutes range. Also, following software engineering practices, we contend than any specification significantly greater than 50 should be decomposed into

modules. This allows additional optimizations of the verifier (in the spirit of [FQ03]), which can take advantage of the additional independence between different modules.

Also, comparing the running times of the different algorithms we can conclude that correctly verifying specifications with unary keys and arithmetic does not introduce a significant performance penalty. This follows from two reasons. First, the worst case upper bound does not hit in the business process specifications used in our experiments. Second, the added complexity of handling keys or arithmetic is offset by the additional pruning happening on the state space.

We contend that the fact that the specifications we generated do not exhibit the worst case behavior is compatible with the intuition that business processes are not used to perform computations on data attributes, which causes the worst case upper bound; but use data attributes in order to control the flow of activities to be performed.

In order to compare our technique with previous techniques for automatic data-aware verification, we translated a web-site specification used in the experiments for the WAVE verifier [DMS$^+$05]. The web-site modeled a travel agency website (e.g. Expedia, Travelocity) and included 20 pages and a database with more than 10 tables of up to 10 in arity. The web-site did not include arithmetic or key constraints. Since we did not have access to the actual experiments run in [DMS$^+$05], we verified properties that implemented the same patterns as the one described in [DMS$^+$05]. The running time spanned a range of hundreds of hundreds of msec to 8 seconds for the mixed algorithm. These are nearly the same running times as reported by [DMS$^+$05], which reports hundreds of msec to 4 seconds, although on a different machine. We argue then that our technique, despite supporting more expressive specifications, does not perform significantly worse than WAVE, that in [DMS$^+$05] compared favorably with traditional non-data-aware techniques like SPIN.

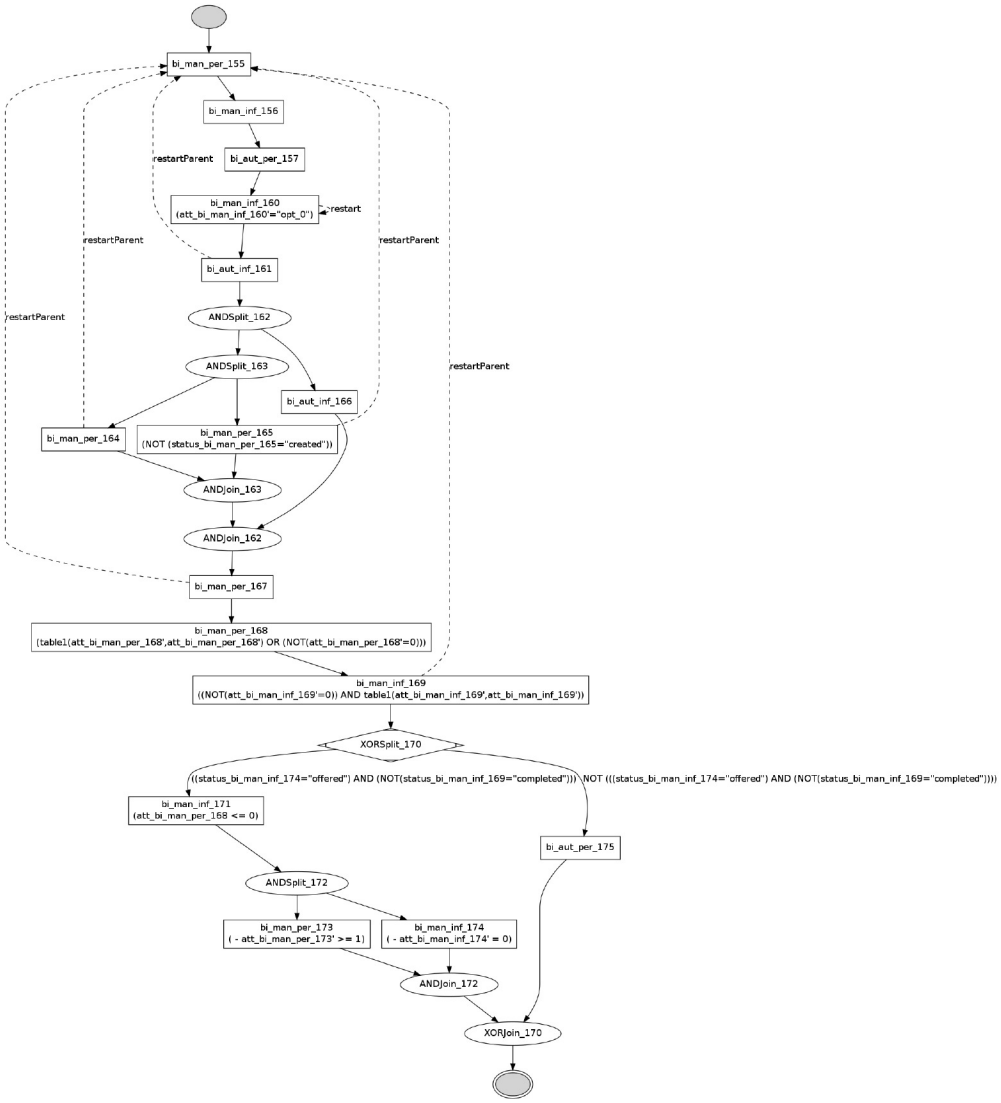

Figure 7.1: Example of a randomly generated AWE

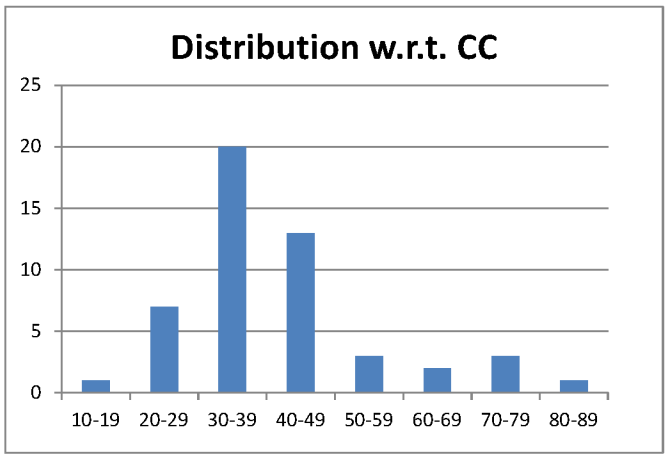

Figure 7.2: Distribution of generated specifications w.r.t. Cyclomatic Complexity

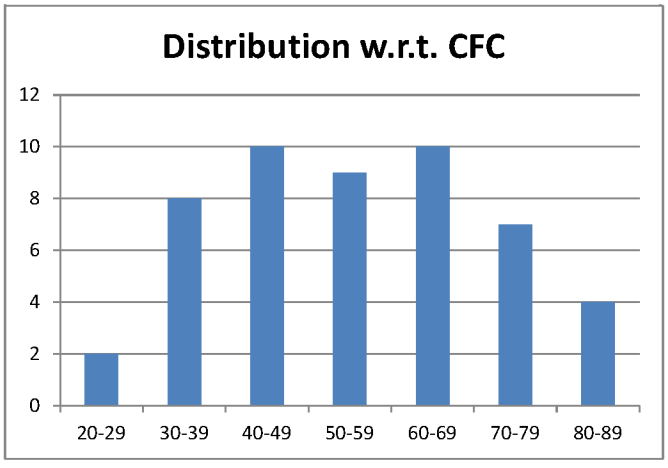

Figure 7.3: Distribution of generated specifications w.r.t. Control Flow Complexity

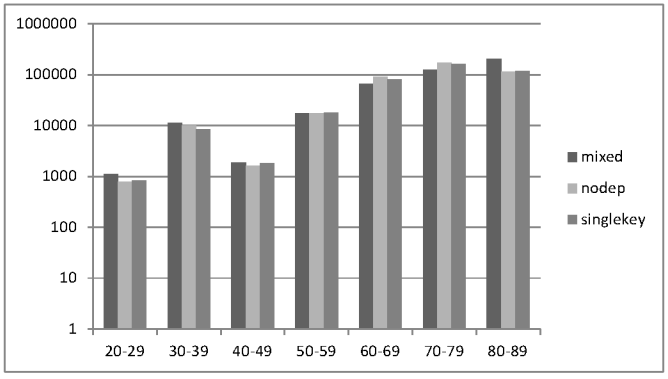

Figure 7.4: Average running time of verification algorithm w.r.t. Control Flow Complexity

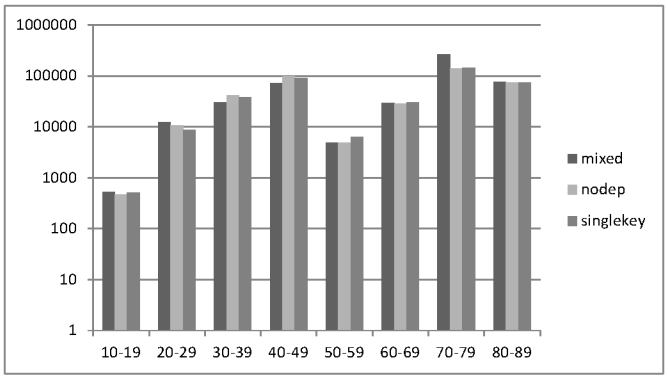

Figure 7.5: Average running time of verification algorithm w.r.t. Cyclomatic Complexity

8 Conclusion

In this thesis we studied the feasibility of automatic verification of temporal properties on business process specifications with data dependencies and arithmetic. Compared to previous works in the verification field (see Section8.1), this thesis focuses on the following aspects:

No loss of expressiveness Real-world business process specifications use both data-aware constructs (with rich data dependencies) and arithmetic operations. We want to be able to perform verification on all the possible behaviors arising from such expressive power. This is in contrast to many previous approaches that abstracts data and arithmetic; and to the ones might maitain the expressiveness but require an expert user to provide additional information to correctly handle data.

Soundness and completeness We want to maintain completeness in our verification technique. This means that we want to avoid false negatives as many application scenarios (such as business rules or policy compliance) can greatly benefit from sound and complete verification.

Automatic verification The technique does not have to rely on an external expert user to help during verification. We want our technique to be amenable to integration in higher-level tools (e.g. business process development tools, monitoring, discovery), this implies that users might be completely unaware of a verification problem being solved under the hood.

Verification has been an important research topic for decades. Verifying general purpose software, however, poses challenges that precludes most of the goals mentioned above. Specifically, general purpose software makes extensive use of data, arithmetic and general recursive control structures. This is required by the generality of

the program behaviors required by implementations in general purpose programming languages.

We contend in this thesis that business processes provide the perfect opportunity to aim for the above mention goals. Work on patterns found in real-world business processes [VDATHKB03, AM00] support the thesis that the recursive structures of business process workflows is different, and more limited, than the one of general programs. We argue that the work in this thesis develops strong foundations for automatic verification of highly expressive business process specifications that is both sound and complete.

Business process model

We propose a syntactic restriction on the recursive structure of business process specifications and termporal properties, called feedback-freedom, that guarantees decidability of the verification problem (Chapters 2 and **??**). The intuititon behind feedback-freedom is reconducible to ideas expressed in studies of real-world business process specifications [AM00]. Also, analogously to cyclomatic complexity [McC76], feedback-freedom might be useful as a theoretically-based design guideline for business processes.

One important feature is missing from our development: support for multiple artifact instances. A first observation, is that any bounded number of instances is directly supported by our model. This case is more useful in business processes as it is for general purpose programs as reported in [VDATHKB03]. Secondly, we want to note that many solutions have been developed for the same problem in standard model checking [FQ03], that we think could be applied to our case. This is a very intersting area for future work.

Complexity and implementation

Even considering the fact tha traditional model checking is PSPACE-complete, the complexity upper bounds identified for our techniques are very high (EXPSPACE for unary keys, and hyperexponential w.r.t. the size of the maximum cluster of attributes involved in arithmetic, Chapter 4). We indentify common classes of specifications for which, given a fixed-arity database, verification of unary keys is PSPACE. Moreover, we

contend that the hyperexponentiality does not manifest in real-world business process specifications as it is the result of a specification using the recursive control to perform arithmetic computations in the artifact attributes. This is compatible with the common understanding of a business process specifications as the *control* of a workflow: the business process describes and/or orchestrates the work performed by the organization.

The set of experiments that we ran on our verifier prototype (Chapters 6 and 7) support our thesis that these techniques performs adequately (i.e. running times in the range of seconds to minutes) for a wide variety of complexities of business process specifications. The complexity class of the specification studied in our experiments is large enough to contain real-world business processes, i.e. cyclomatic complexity 80+ and control flow complexity 90+ (as per design guidelines [McC76, Car05]). Design guidelines support our thesis that specification larger than that should be modularized, enabling the exploitation of additional verification techniques such as compositional model checking [BCC98]. Specific techniques could take advantage of higher level module information not present in our specification language. For instance, modules could declare data attributes required in read-mode and write-mode, decoupling different portions of the specification, which results in the pruning of execution paths to be checked by the verifier.

Concluding, we believe that the work in this thesis proves the feasibility of automatic verification of temporal properties on business process specifications with data dependencies and arithmetic. The rest of this chapter summarizes some of the related work.

8.1 Related Work

Data-aware business process models The specific notion of artifact was first introduced in [NC03] and was further studied, from both practical and theoretical perspectives, in [BCK$^+$07, B$^+$05, GBS07, GS07, BGH$^+$07, LBW07, KLW08, KRG07, ea10]. Some key roots of the artifact model are present in "adaptive objects"[KNH$^+$03], "adaptive business objects" [NK05], "business entities", "document-driven" workflow

[WK05] and "document" engineering [GM05]. The Vortex framework [HLS+99, DHK+99, HLK+00] also allows the specification of database manipulations and provides declarative specifications for when services are applicable to a given artifact. The artifact model considered here is closely related to that of semantic web services in general. In particular, the OWL-S proposal [MSZ01, M+03] describes the semantics of services with input, output, pre- and post-conditions.

Static analysis of data-aware business processes Work on formal analysis of artifact-based business processes in restricted contexts has been reported in [GBS07, GS07, BGH+07]. Properties investigated include reachability [GBS07, GS07], general temporal constraints [GS07], and the existence of complete execution or dead end [BGH+07]. Citations [GBS07, GS07] are focused on an essentially procedural version of artifact-centric business processes, and [BGH+07] is the first to study a declarative version. For the variants considered in each paper, verification is generally undecidable; decidability results were obtained only under rather severe restrictions, e.g., restricting all pre-conditions to be "true" [GBS07], restricting to bounded domains [GS07, BGH+07], or restricting the pre- and post-conditions to refer only to artifacts (and not their variable values) [GS07]. None of the above papers permit an underlying database, integrity constraints, or arithmetic.

[CGHS09] adopts an artifact model variation with arithmetic operations but no database (and therefore no integrity constraints). It proposes a criterion for comparing the expressiveness of specifications using the notion of *dominance*, based on the input/output pairs of business processes. Decidability is shown only by restricting runs to bounded length. [ZSYQ09] addresses the problem of the existence of a run that satisfies a temporal property, for a restricted case with no database, no arithmetic, and only propositional LTL properties. [BHCDG+11] considers another variety of the artifact model, with database but no arithmetic and no data dependencies, and limited modeling of the input from the environment. The work focuses on static verification of properties in a very powerful language (first order μ-calculus) which subsumes the temporal logic we consider (first order LTL), in particular allowing branching time. This expressivity comes at the cost of restricting verification decidability to the case when the initial database contents are given. In contrast, we verify correctness properties for all possible

initializations of the database.

Static analysis for semantic web services is considered in [NM02], but in a context restricted to finite domains.

More recently, [ASV09] has studied automatic verification in the context of business processes based on Active XML documents.

The work in this thesis is most closely related to the one in [DHPV09], which identifies the class of *guarded* artifact systems and LTL-FO properties, for which verification is decidable. The two settings have in common the underlying database and the infinite data domain with a dense linear order, as well as the syntax for pre-, post-conditions, and properties. However, [DHPV09] allows artifacts to contain relations (called "state relations") in addition to the record of variables, but considers no dependencies and no arithmetic operations. Our previous results do not apply in the new context, as Theorem 2.2.4 shows undecidability even when adding to a guarded artifact system a single functional dependency, or alternately, when the only allowed arithmetic operation consists in incrementing counters. The novel proof technique based on describing configuratons using inherited constraints is fundamentally different from the one employed for guardedness. It is precisely the use of inherited constraints that enables the support of dependencies (by chasing the inherited constraints with them) as well as any decidable interpretation of $\mathscr{C}$ (by splitting the inherited constraints over the two schemas $\mathscr{C}$ and $\mathscr{DB}$ and solving the $\mathscr{C}$-satisfiability sub-problem in isolation).

The works [DSV07, Spi03, AVFY00] are ancestors of [DHPV09] from the context of verification of electronic commerce applications. Their models could conceptually (if not naturally) be encoded as artifact systems, but they correspond only to particular cases of the model in [DHPV09]. They all disallow the linear order on the domain. Also, limit artifact values to essentially come from the active domain of the database, thus ruling out external inputs, partially-specified services, and arithmetic.

Infinite-state systems We expect our results to be of interest to the verification community at large, since artifact systems are a particular case of infinite-state systems. Research on automatic verification of infinite-state systems has recently focused on extending classical model checking techniques (e.g., see [BCMS01] for a survey). However, in much of this work the emphasis is on studying recursive control rather than

data, which is either ignored or finitely abstracted. More recent work has been focusing specifically on data as a source of infinity. This includes augmenting recursive procedures with integer parameters [BHM03], rewriting systems with data [BJS07, BHJS07], Petri nets with data associated to tokens [LNO$^+$07], automata and logics over infinite alphabets [BPT03, Bou02, NSV04, DL06, JL07, BMS$^+$06, BHJS07], and temporal logics manipulating data [DL06, DLS08]. However, the restricted use of data and the particular properties verified have limited applicability to the business artifacts setting.

Acknowledgement

Alin Deutsch and Victor Vianu co-authored this chapter.

Bibliography

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[AM00] Alessandra Agostini and Giorgio De Michelis. Improving flexibility of workflow management systems. In *Business Process Management*, pages 218–234, 2000.

[ant] Antlr. http://www.antlr.org/.

[ASV09] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34(4), 2009.

[AVFY00] S. Abiteboul, V. Vianu, B.S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000. Extended abstract in PODS 98.

[B+05] K. Bhattacharya et al. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems Journal*, 44(1):145–162, 2005.

[BCC98] Sergey Berezin, SÃl'rgio Campos, and Edmund Clarke. Compositional reasoning in model checking. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer Berlin / Heidelberg, 1998.

[BCK+07] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.

[BCMS01] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification of infinite structures. In *Handbook of Process Algebra*, pages 545–623. Elsevier Science, 2001.

[BGH+07] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *Proc.*

Int. Conf. on Business Process Management (BPM), pages 288–304, 2007.

[BHCDG+11] Babak Bagheri-Hariri, Diego Calvanese, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Foundations of relational artifacts verification. In *Proc. of 9th Int. Conference on Business Process Management (BPM 2011)*, 2011.

[BHJS07] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *FCT'07*, volume 4639 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[BHM03] A. Bouajjani, P. Habermehl, and R. Mayr. Automatic verification of recursive procedures with one integer parameter. *Theoretical Computer Science*, 295:85–106, 2003.

[Bir96] J-C Birget. Two-way automata and length-preserving homomorphisms. *Theory of Computing Systems*, 29(3):191–226, 1996.

[BJS07] A. Bouajjani, Y. Jurski, and M. Sighireanu. A generic framework for reasoning about dynamic networks of infinite-state processes. In *TACAS'07*, volume 4424 of *Lecture Notes in Computer Science*, pages 690–705. Springer, 2007.

[BMS+06] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.

[Bou02] P. Bouyer. A logical characterization of data languages. *Information Processing Letters*, 84(2):75–85, 2002.

[BPT03] P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2):137–162, 2003.

[Car05] Jorge Cardoso. Evaluating the process control-flow complexity measure. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 803–804, Washington, DC, USA, 2005. IEEE Computer Society.

[CGHS09] Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Jianwen Su. Artifact-centric workflow dominance. In *ICSOC/ServiceWave*, pages 130–143, 2009.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Workshop on Formal Methods in Software Practice*, 1998.

[DBL01] *Proceedings of GROUP 2001, ACM 2001 International Conference on Supporting Group Work, September 30 - October 3, 2001, Boulder, Colorado, USA*. ACM, 2001.

[DDV11] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. In *International Conference on Database Theory (ICDT)*, 2011.

[DHK+99] G. Dong, R. Hull, B. Kumar, J Su, and G Zhou. A framework for optimizing distributed workflow executions. In *Proc. Intl. Workshop on Database Programming Languages (DBPL)*, pages 152–167, 1999.

[DHPV09] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267, 2009.

[dig] Apache digester. http://commons.apache.org/digester/.

[DL06] Stéphane Demri and Ranko Lazić. LTL with the Freeze Quantifier and Register Automata. In *LICS*, pages 17–26, 2006.

[DLS08] Stéphane Demri, Ranko Lazić, and Arnaud Sangnier. Model checking freeze LTL over one-counter automata. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'08)*, pages 490–504, 2008.

[DMS+05] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 539–550, New York, NY, USA, 2005. ACM.

[DNR08] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.

[DSV07] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web applications. *JCSS*, 73(3):442–474, 2007.

[ea10] R. Hull et al. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *Proc. of 7th Intl. Workshop on Web Services and Formal Methods (WS-FM)*, 2010.

[Fag82] Ronald Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.

[FKMP03] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In Thomas Ball and Sriram Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 624–624. Springer Berlin / Heidelberg, 2003.

[GBS07] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *IEEE International Conference on Service-Oriented Computing and Applications*, 2007.

[GL06] Volker Gruhn and Ralf Laue. Complexity metrics for business process models. In *9th international conference on business information systems (BIS 2006), volume 85 of Lecture Notes in Informatics*, pages 1–12, 2006.

[GM05] R.J. Glushko and T. McGrath. *Document Engineering: Analyzing and Designing Documents for Business Informatics and Web Services*. MIT Press, Cmabridge, MA, 2005.

[GS07] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *Proceedings of 5th International Conference on Service-Oriented Computing (ICSOC)*, Vienna, Austria, September 2007.

[HLK+00] R. Hull, F. Llirbat, B. Kumar, G. Zhou, G. Dong, and J. Su. Optimization techniques for data-intensive decision flows. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 281–292, 2000.

[HLS+99] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration*, 1999.

[JL07] Marcin Jurdzinski and Ranko Lazić. Alternation-free modal mu-calculus for data trees. In *LICS*, pages 131–140, 2007.

[jpf] Java pathfinder. http://javapathfinder.sourceforge.net/.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *STOC*, pages 302–311, 1984.

[KLW08] S. Kumaran, R. Liu, and F. Y. Wu. On the duality of information-centric and activity-centric models of business processes. In *Proc. Intl. Conf. on Advanced Information Systems Engineering (CAISE)*, 2008.

[KNH+03] S. Kumaran, P. Nandi, T. Heath, K. Bhaskaran, and R. Das. ADoc-oriented programming. In *Symp. on Applications and the Internet (SAINT)*, pages 334–343, 2003.

[KRG07] J. Küster, K. Ryndina, and H. Gall. Generation of BPM for object life cycle compliance. In *Proceedings of 5th International Conference on Business Process Management (BPM)*, 2007.

[LBW07] R. Liu, K. Bhattacharya, and F. Y. Wu. Modeling business contexture and behavior using business artifacts. In *CAiSE*, volume 4495 of *LNCS*, 2007.

[Lib04] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[LLS84] Richard E. Ladner, Richard J. Lipton, and Larry J. Stockmeyer. Alternating pushdown and stack automata. *SIAM J. Comput.*, 13(1):135–155, 1984.

[LNO+07] R. Lazić, Th. Newcomb, J. Ouaknine, A. Roscoe, and J. Worrell. Nets with tokens which carry data. In *ICATPN'07*, volume 4546 of *Lecture Notes in Computer Science*, pages 301–320. Springer, 2007.

[lps] lp_solve. http://sourceforge.net/projects/lpsolve/.

[LS11] L.Segoufin and S.Torunczyk. Automata based verification over linearly ordered data domains. In *Int'l. Symp. on Theoretical Aspects of Computer Science (STACS)*, 2011.

[M+03] D. Martin et al. OWL-S: Semantic markup for web services, W3C Member Submission, November 2003.

[McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[Min67] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[MNG08] Wim Martens, Frank Neven, and Marc Gyssens. Typechecking top-down XML transformations: Fixed input or output schemas. *Inf. Comput.*, 206(7):806–827, 2008. Preliminary version in ICDT 2003.

[MSWL10] Michael Meier, Michael Schmidt, Fang Wei, and Georg Lausen. Semantic query optimization in the presence of types. In *PODS*, pages 111–122, 2010.

[MSZ01] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[NC03] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[NK05] P. Nandi and S. Kumaran. Adaptive business objects – a new component model for business integration. In *Proc. Intl. Conf. on Enterprise Information Systems*, pages 179–188, 2005.

[NM02] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Intl. World Wide Web Conf. (WWW2002)*, 2002.

[NSV04] F. Neven, T. Schwentick, and V. Vianu. Finite State Machines for Strings Over Infinite Alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[RHE05] Nick Russell, Arthur H. M. Ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSEâĂŹ05), volume 3520 of Lecture Notes in Computer Science*, pages 216–232. Springer, 2005.

[RtHEvdA05] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In *IN âĂŸPROCEEDINGS OF THE 25TH INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING (ERâĂŹ2005*. Springer, 2005.

[RvdAtH] Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. Workflow exception patterns. In *Proceedings of 18th CAiSE*, pages 288–302.

[Spi03] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS.*, 66(1):40–65, 2003. Extended abstract in PODS 2000.

[TLR07] L. H. Thom, C. Lochpe, and M. U. Reichert. Workflow patterns for business process modeling. In *Proceedings of Workshops and Doctoral Consortium of the 19th International Conference on Advanced Information Systems Engineering (CAiSE 2007), Trondheim, Norway*, volume I, pages 349–358, Norway, June 2007. Tapir Academic Press.

[VDATHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14:5–51, July 2003.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, 1986.

[WK05] J. Wang and A. Kumar. A framework for document-driven workflow systems. In *Business Process Management*, pages 285–301, 2005.

[ZSYQ09] Xiangpeng Zhao, Jianwen Su, Hongli Yang, and Zongyan Qiu. Enforcing constraints on life cycles of business artifacts. In *TASE*, pages 111–118, 2009.