Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVIII Ciclo – 2005

Formal Verification: further Complexity Issues and Applications

Andrea Ferrara

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVIII Ciclo - 2005

Andrea Ferrara

Formal Verification: further Complexity Issues and Applications

Thesis Committee

Marco Schaerf (Advisor)
Giuseppe De Giacomo
Francesco Quaglia

Reviewers

Enrico Giunchiglia
Paolo Traverso

Author's address:
Andrea Ferrara
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: ferrara@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼ferrara/

To F.F.

Acknowledgements

I would like to thank Marco Schaerf and Paolo Liberatore, for their continuous support and valuable suggestions, that have made my research possible. The many hours of discussions have been a very pleasure for me, and have introduced me to research and to fascinating issues.

I would also like to thank Moshe Y. Vardi for all he teaches to me, and Guoqiang Pan and Stefano Tonetta for their help and assistance in difficult situations I had when I was in Houston.

I wish to thank Giuseppe De Giacomo for his interesting suggestions and discussions. I also wish to thank the external reviewers, Paolo Traverso and Enrico Giunchiglia for their advises about the contents and presentation of this thesis.

Many thanks to all the people of the Dipartimento di Informatica e Sistemistica of the Università di Roma "La Sapienza", specifically Marco Cadoli, Massimo Mecella, Daniela Berardi, and Toni Mancini.

The last, but not least, words are dedicated to my father and mother, Renato and Franca, and to my sweet sister, Francesca. Without their support, patience and love this work would have never come into existence.

Finally, an "in love" thank to Floriana, always been near me in these last years of hard work and study.

Roma, Italy — Andrea Ferrara
December 15th, 2005

Contents

Chapter 1

Introduction

1.1 Contents

Formal verification of discrete systems is a verification method in which a system, often called a model, is described by the possible transitions of its components. Using this description, we can verify whether:

the system satisfies properties encoded in a temporal modal logic [101]. This problem is called *Temporal Logic Model Checking* [39] (Model Checking for short). Examples of these properties are:

does the system reach certain states? (Reachability)

does the system show a legal behavior? (Safety)

the behaviors of a system is included in the behaviors of another system. Depending the definition of this behavioral relation, we have a corresponding problem: for instance *Simulation*, *Trace Containment* [17, 89, 88].

Formal verification has many industrial applications. It is used, for example, for the verification of protocols and hardware circuits [59, 14]. Many tools, called *model checkers*, have been developed to this aim. The most famous ones are SPIN [67] and SMV [86] (with its many incarnations: NuSMV [37], RuleBase [15]), VIS [29], and FormalCheck [65].

The complexity of Model Checking (PSPACE-complete [75]), Simulation (EXPTIME-complete [64]), Containment (EXPSPACE-complete [64]) is known. In this thesis, we first deal with the complexity of these and other problems under the hypothesis of structural restrictions (e.g. treewidth [103, 102, 107]). With these structural restrictions, several PSPACE-complete and NP-complete problems becomes PTIME (e.g. CNF-SAT) (see [27, 26, 25] for an overview).

We first show a positive result: the complexity of image computation operation decreases. The image computation operation is a basic operation in the explicit Model Checking, one of most effective approach to this problem [39]. Then, we present a negative result about Bounded Model Checking (BMC for short) [22], a SAT-based Model Checking in which the behaviors of the system are unrolled in a Boolean formula (that exactly describes them); we show that the unrolling in the Bounded Model Checking does not preserve the treewidth, therefore the verification of a system using the BMC approach cannot take an advantage from structural restrictions such as treewidth. We continue proving a sequence of negative results: the complexity of Simulation, Containment and Model Checking does not decrease under the hypothesis of bounded treewidth on the model.

Then, we study whether there is another restriction that can help in solving Model Checking; we assume that a part of the input (either the model or the formula) can be preprocessed. In many

cases, the two inputs of the model checking problem (the model and the formula) can be processed in a different way. If we want to verify several properties of the same system, it makes sense to spend more time on the model alone, if the verification of the properties becomes faster. Many tools allow to build the model separately from checking the formula [36, 114, 69]; in this way, one can reuse the same model, compiled into a data structure, in order to check several formulae.

In the same way, we may wish to verify the same property on different systems: the property is this time the part we can spend more time on. Many tools allow populating a property database [36, 114, 69], i.e., a collection of temporal formulae which will be checked on the models. We imagine a situation in which we early establish the requirements that our system must satisfy, even before the system is actually designed. As a result, and we can fill a database of temporal formulae, but we do not yet describe the system. While the design/modeling of the system goes on, we can preprocess the formulae (without knowledge of the model, which is not yet known). Whenever the system is specified, we can then use the result of this preprocessing step to check the model against the formulae.

We prove that preprocessing the model or the property using any amount of time and storing the result of this preprocessing in a polynomial-space data structure, does not decrease the worst case complexity. As a side result, we prove a theorem about the size increase in the worst case of a large class of data structures adopted in Symbolic Model Checking algorithms. Symbolic Model Checking algorithms are a large class of algorithms for solving the Model Checking problem; they work by manipulating sets of states, and these sets are often represented by BDDs or other data structures [43]. It has been observed that the size of BDDs may grow exponentially as the model and formula increase in size. We *formally* prove that a superpolynomial increase of the size of these BDDs is unavoidable in the worst case. While this superpolynomial growth has been empirically observed, and it is proved for particular problems (e.g. integer multiplication [31]), to the best of our knowledge, it has never been proved in the general case so far. This result not only holds for all types of BDDs regardless of the variable ordering, but also for more powerful data structures, such as BEDs, RBCs, MTBDDs, and ADDs.

Then, we discuss some problems related to the reachability property, that is interesting also in the field of reasoning about actions. In particular, we consider the planning area, where the reachability problem is equivalent to the existence of a combination of actions that reaches a given goal (the property). The connection between Model Checking and Planning are discussed and used in [58, 18, 20, 77, 98, 21, 19]. In details, we show the practical importance to decide whether an action is redundant, i.e. it is not needed to reach the goal. Then, we study the computational complexity of several problems related to the redundancy of actions: checking whether a domain contains a redundant action, what is the minimal number of actions needed to make the goal reachable, checking whether the removal of an action does not increase the minimal plan length, and other related problems.

Finally, we present an application of formal verification to Web services (WS for short). Our aim is to reduce the effort of testing phase in the WS development. We present a framework for the design and the verification of WSs using process algebras [17] and their tools. Process algebras are an algebraic formalisms whose sematics is based on transition systems. We define a two-way mapping between abstract specifications written using these calculi and executable Web services written in BPEL [4, 1, 2]; the translation includes also compensation, event, and fault handlers. The following choices are available: design and verification in BPEL, using process algebra tools, or design and verification in process algebra and automatically obtaining the corresponding BPEL code. The approaches can be combined. Process algebras are not useful only for temporal logic model checking: we remark the use of simulation/bisimulation for verification, for the hierarchical

refinement design method, for the service redundancy analysis in a community of Web services, and for replacing a service with another one in a composition.

1.2 Main Results

The main results of the thesis are:

about the complexity of formal verification problems, under the hypothesis of structural restrictions the following results hold (Chapter 3):

the *Image Computation* complexity on a transition system with $|V|$ states and treewidth at most k is $O(|V|^2 \cdot k)$; in the case of unbounded treewidth is $O(|V|^3)$. This result is in Section 3.2.

Bounded Model Checking Unrolling does not preserve treewidth (Section 3.3)

Temporal Logic Model Checking is PSPACE-complete (Section 3.4)

Simulation is EXPTIME-complete (Section 3.4)

Containment is EXPSPACE-complete (Section 3.4)

about the complexity of preprocessing Model Checking and BDDs superpolynomial growth (Chapter 4)

about the complexity of checking action redundancy, we study some problems related with reachability (Chapter 5)

about the application of the formal verification to Web services area, we present a two-way mapping between process algebra e BPEL (Chapter 6)

Chapter 2

Preliminaries

In this chapter, we introduce the basic definitions needed in the thesis.

2.1 Treewidth and Formal Verification

2.1.1 Treewidth

In this section we recall the basic definitions about treewidth that are needed in Chapter 3. The notions of treewidth and pathwidth were introduced in [103, 102].

Definition 1 *A tree decomposition of a graph $G = (V, E)$ is a pair (T, X) with $T = I, F$ is a tree whose I is the set of nodes and F the set of edges and $X = \{X_i | i \in I\}$ a family of subset of V, one for each node of T. T is such that*

$\bigcup_{i \in I} X_i = V$.

for all edges $(v, w) \in E$, there exists an $i \in I$ with $v \in X$, and $w \in X_i$.

for all $i, j, k \in I$: if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The *width* of a tree decomposition (T, X) is $max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G is the minimum width over all possible tree decompositions of G.

One obtain an equivalent definition, when the third condition in the definition of tree decomposition is replaced by:

For all $v \in V$, the set of nodes $i \in I | v \in X_i$ forms a connected component part (i.e. a subtree) of T.

The notion of path decomposition restrict the tree to be a path.

Definition 2 *A path decomposition of a graph $G = (V, E)$ is a pair (T, X) with $T = I, F$ is a path whose I is the set of nodes and F the set of edges and $X = \{X_i | i \in I\}$ a family of subset of V, one for each node of T. T is such that*

$\bigcup_{i \in I} X_i = V$.

for all edges $(v, w) \in E$, there exists an $i \in I$ with $v \in X$, and $w \in X_i$.

for all $i, j, k \in I$: if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The *width* of a path decomposition (T, X) is $maxi \in I|X_i| - 1$. The *pathwidth* of a graph G is the minimum width over all possible path decompositions of G.

One obtain an equivalent definition, when the third condition in the definition of path decomposition is replaced by:

For all $v \in V$, the set of nodes $i \in I|v \in X_i$ forms a connected component part (i.e. two adiacent nodes) of T.

By Corollary 24 in [27], we know that for a graph G with n vertices we have that $pathwidth(G) = O(treewidth(G) \cdot \log n)$. Clearly, $treewidth(G) \leq pathwidth(G)$.

Definition 3 *The Gaifman graph of a CNF formula is a graph having one vertex for each variable and an edge (v_1, v_2) if the variables v_1 and v_2 occur in the same clause of the formula. By treewidth (pathwidth) of a CNF formula we refer to the treewidth (pathwidth) of its Gaifman graph.*

2.1.2 Transition Systems

Now, we recall the basic definitions about Transition Systems that are needed in Chapter 3. We introduce the definitions about the non deterministic transition system with bounded concurrency. A non deterministic transition system with bounded concurrency (*concurrent transition system* for short) is a tuple $P = \langle O, P_1, \ldots, P_n \rangle$ consisting of a finite set O of *observable events* and n *components* $P_1, \ldots, P_n$ for some $n \geq 1$. Each component P_i is a tuple $\langle O_i, W_i, W_i^0, \delta_i, L_i \rangle$, where:

$O_i \subseteq O$ is a set of local observable events. The O_j are not necessarily pairwise disjoint; hence, observable events may be shared by several components. We require that $\bigcup_{j \in I}^{n} O_j$

W_i is a finite set of states, and we require that the W_j be pairwise disjoint. Also we let $W = \bigcup_{j \in I}^{n} W_j$

$W_i^0 \subseteq W_i$ is the set of initial states.

$\delta \subseteq W_i \times \beta(W) \times W_i$ is a transition relation, where $\beta(W)$ denotes the set of all Boolean propositional formulae over W.

$L_i : W_i \rightarrow 2^{O_i}$ is a labeling function that labels each state with a set of local observable events. The intuition is that for each state $w \in W_i$, $L_i(w)$ are the events that occur, or hold, in w

Since states are labeled with sets of elements from O, we refer to $\Sigma = 2^O$ as the *alphabet* of P. While each component of P has its local observable events and its own states and transitions, these transitions depend not only on the component's current state but also on the current states of the other components. Also, as we shall now see, the labels of the components are required to agree on shared observable events.

A *configuration* of P is a tuple $c = \langle w_1, w_2, \ldots, w_n, \sigma \rangle \in W_1 \times W_2 \times \cdots \times W_n \times \Sigma$, satisfying $L_i(w_i) = \sigma \cap O_i$ for all $1 \leq i \leq n$. Thus, a configuration describes the current state of each of the components, as well as the set of observable events labeling these states. The requirement on σ implies that these labels are *consistent*, i.e., for any P_i and P_j, and for each $o \in O_i \cap O_j$, either $o \in L_i(w_i) \cap L_j(w_j)$ (in which case, $o \in \sigma$), or $o \notin L_i(w_i) \cup L_j(w_j)$ (in which case, $o \notin \sigma$). For a configuration $c = \langle w_1, w_2, \ldots, w_n, \sigma \rangle$, we term $\langle w_1, w_2, \ldots, w_n \rangle$ the *global state* of c, and we term σ the *label* of c, and denote it by $L(c)$. A configuration is *initial* if for all $1 \leq i \leq n$, we have $w_i \in W_i^0$. We use C to denote the set of all configurations of a given system P, and C_0 to denote the set of all its initial configurations. We also use $c[i]$ to refer to P_i's state in c.

For a propositional formula θ in $\mathcal{B}(W)$ and a global state $p = \langle w_1, w_2, \ldots, w_n \rangle$, we say that p *satisfies* θ if assigning **true** to states in p and **false** to states not in p makes θ true. For example, $s_1 \wedge (t_1 \vee t_2)$, with $s_1 \in W_1$ and $\{t_1, t_2\} \subseteq W_2$, is satisfied by every global state in which P_1 is in state s_1 and P_2 is in either t_1 or t_2. We shall sometimes write disjunctions as sets, so that the above formula can be written $\{s_1\} \wedge \{t_1, t_2\}$. Formulas in $\mathcal{B}(W)$ that appear in transitions are called *conditions*.

Given two configurations $c = \langle w_1, w_2, \ldots, w_n, \sigma \rangle$ and $c' = \langle w'_1, w'_2, \ldots, w'_n, \sigma' \rangle$, we say that c' is a *successor of* c *in* P, and write $succ_P(c, c')$, if for all $1 \leq i \leq n$ there is $\langle w_i, \theta_i, w'_i \rangle \in \delta_i$ such that $\langle w_1, w_2, \ldots, w_n \rangle$ satisfies θ_i. In other words, a successor configuration is obtained by simultaneously applying to all the components a transition that is enabled in the current configuration. Note that by requiring that successors are indeed configurations, we are saying that transitions can only lead to states satisfying the consistency criterion, to the effect that they agree on the labels for shared observable events. [1]

Given a configuration c, a *c-computation* of P is an infinite sequence $\pi = c_0, c_1, \ldots$ of configurations, such that $c_0 = c$ and for all $i \geq 0$ we have $succ_P(c_i, c_{i+1})$. A *computation* of P is a c-computation for some $c \in C_0$. The computation $c_0, c_1, \ldots$ *generates* the infinite *trace* $\rho \in \Sigma^\omega$, defined by $\rho = L(c_0) \cdot L(c_1) \cdots$.

We use $\mathcal{T}(P^c)$ to denote the set of all traces generated by c-computations, and the *trace set* $\mathcal{T}(P)$ of P is then defined as $\bigcup_{c \in C_0} \mathcal{T}(P^c)$. In this way, each concurrent transition system P defines a subset of Σ^ω. We say that P *accepts* a trace ρ if $\rho \in \mathcal{T}(P)$. Also, we say that P is *empty* if $\mathcal{T}(P) = \emptyset$; i.e., P has no computation, and that P is *universal* if $\mathcal{T}(P) = \Sigma^\omega$; i.e., every trace in Σ^ω is generated by some fair computation of P.

The *size* of a concurrent transition system P is the sum of the sizes of its components. Symbolically, $|P| = |P_1| + \cdots + |P_n|$. Here, for a component $P_i = \langle O_i, W_i, W_i^0, \delta_i, L_i, \alpha_i \rangle$, we define $|P_i| = |O_i| + |W_i| + |\delta_i| + |L_i| + |\alpha_i|$, where $|\delta_i| = \sum_{\langle w, \theta, w' \rangle \in \delta_i} |\theta|$, $|L_i| = |O_i| \cdot |W_i|$, and $|\alpha_i|$ is the sum of the cardinalities of the sets in α_i. Clearly, P can be stored in space $O(|P|)$.

When P has a single component, we say that it is a *sequential transition system*. Note that the transition relation of a sequential transition system can be really viewed as a subset of $W \times W$, and that a configuration of a sequential transition system is simply a labeled state.

Treewidth and Transition Systems Now, we introduce the definitions about the *local* and *global* treewidth, and the degree of a graph.

Definition 4 *The communication graph of a concurrent transition system P is a graph having one vertex for each component and an edge (v_i, v_j) if either the component for v_i and the component for v_j share observable events or if the transition relation of one of the components for v_i or v_j refer to the variables of the other.*

Definition 5 *The local treewidth of the concurrent transition system P is the treewidth of its communication graph.*

By the Theorem 2.2 in [64], every concurrent transition system P can be translated into a sequential transition system of size $2^{O(|P|)}$.

Definition 6 *The global treewidth of the concurrent transition system P is the treewidth of its equivalent sequential transition system.*

[1] This requirement could obviously have been imposed implicitly in the transition relation, by disallowing in the δ_i any tuples $\langle w, \theta, w' \rangle$ for which θ holds when the states of some of the components are mutually inconsistent. Since we always want the components to agree on the labeling of shared observable events, we have set up our definitions of configurations and successors to make this requirement explicit. Technically, imposing the requirement in the transition relation could be done by replacing each condition θ by $\theta \wedge \varphi$, where $\varphi \in \mathcal{B}(W)$ is satisfied in a global state exactly when the states of all its components are mutually consistent. The length of φ is linear in $|W|$ and $|O|$, so that the explicit requirement does not involve a substantial decrease in succinctness.

Definition 7 *The degree of a graph is the maximum vertex degree, in other words, the maximum count of arcs connected to a single vertex in the graph.*

A graph with bounded pathwidth and bounded degree has bounded cutwidth [112]. The pathwidth implies many other structural restrictions [107].

Example 1 *We construct a concurrent transition system P to encode a (ripple-carry) binary counter; it can count up to 2^n in base 2 using n components. Each component P_i is used to store the i-th bit (the bit with weight 2^{i-1}), so P_1 is the least significant bit and P_n is the most significant bit. The set of observable events is the bit-value stored by each component, and the counter works by ripple carry propagation.*

Formally, given the number of bits n, P is $\langle\{bit_1,\ldots,bit_n\},P_1,\ldots,P_n\rangle$, where $P_i = \langle\{bit_i\}, \{s^i_{00}, s^i_{01}, s^i_{10}\},\{I^i\}, \delta_i, L_i\rangle$. For each state s^i_{jk}, j represent the carry status, and k represent the bit state, for example, the state s^i_{10} represents the case where the value of bit i is 0 and a carry is propagated toward bit $i+1$. I^i is an initial state [2].

In Figure 2.1 we show the P_i process, representing a cell of the counter. The edges are labeled by the condition of the transition relation: c_{i-1} means that the carry of the process P_{i-1} is 1, and it corresponds to s^{i-1}_{10}, $\neg c_{i-1}$ means that the carry of P_i is 0 and it corresponds to $s^{i-1}_{00} \vee s^{i-1}_{01}$.

We remark that P_1 corresponds to the least significant bit of the counter, and the c_0 is always 1. We define δ_i and L_i as follows:

For $i = 1\ldots,n$, $\delta_i = \{\langle s^i_{00}, \neg c_{i-1}, s^i_{00}\rangle,\ \langle s^i_{00}, c_{i-1}, s^i_{01}\rangle,\ \langle s^i_{01}, \neg c_{i-1}, s^i_{01}\rangle,\ \langle s^i_{01}, c_{i-1}, s^i_{10}\rangle, \langle s^i_{10}, \neg c_{i-1}, s^i_{00}\rangle, \langle s^i_{10}, c_{i-1}, s^i_{01}\rangle,\}$.

For every P_i, we set $L_i(s^i_{00}) = L_i(s^i_{10}) = \emptyset$, $L_i(s^i_{01}) = \{bit_i\}$.

The communication graph of this counter have constant pathwidth, since each component P_i interacts only with the components P_{i-1} and P_{i+1}, thus forming a path.

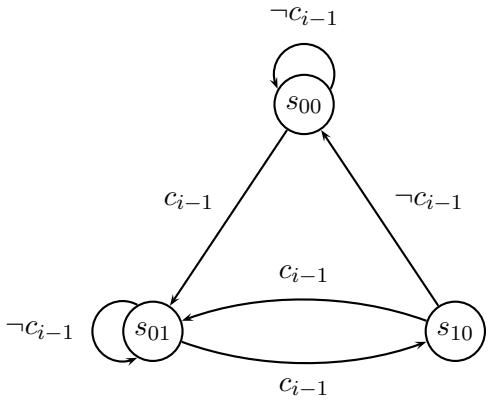

Figure 2.1: A cell of the counter

2.1.3 Verification Problems

In this section, we introduce the definitions for the verification problems that we consider in Chapter 3: model checking, containment, simulation, image computation. Complexity results about model checking are presented also in Chapter 4, where we use, for the sake of the proof simplicity, an alternative and equivalent definition of transition system introduced in Section 2.1.4.

[2] Note if we start with s^i_{00} for all states, the ripple-carry nature of the counter would take $2^n + n - 1$ cycles to flip the carry state of the most significant bit, but we will simply assert there exists an initial sequence such that the counter would count exactly 2^n.

The Temporal Logics and Model Checking The temporal logics [101] often used in the model checking are *CTL* and *LTL*, that are fragment of *CTL**. The logic *CTL** combines both branching-time and linear-time operators [45]. A path quantifier, either A ("for all paths") or E ("for some path"), can prefix an assertion composed of an arbitrary combination of the linear-time operators X ("next time"), and U ("until"). There are two types of formulas in CTL*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific path. Formally, let AP be a set of atomic proposition names. A CTL* state formula is either:

true, **false**, p, or $\neg p$, for all $p \in AP$;

$\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are CTL* state formulas;

$A\psi$ or $E\psi$, where ψ is a CTL* path formula.

a CTL* path formula is either:

a CTL* state formula;

$\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\neg\psi_1$, $X\psi_1$, $\psi_1 U \psi_2$, where ψ_1 and ψ_2 are CTL* path formulas.

CTL* is the set of state formulas generated by the above rules.

The logic *CTL* is a restricted subset of CTL* in which the temporal operators must be immediately preceded by a path quantifier. Formally, it is the subset of CTL* obtained by restricting the path formulas to be $X\varphi_1$, $\varphi_1 U \varphi_2$, or $\varphi_1 \tilde{U} \varphi_2$, where φ_1 and φ_2 are CTL state formulas.

The logic *LTL* is the fragment of CTL* in which the temporal operators does not have path quantifier; only state formulae are allowed.

We use the following abbreviations in writing formulas: $F\psi = \mathbf{true} U \psi$ ("eventually").

The semantics of CTL* is defined with respect to a *Kripke structure* $K = \langle AP, W, R, w_0, L\rangle$, where AP is a set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $\langle w, w'\rangle \in R$), w_0 is an initial state, and $L : W \to 2^{AP}$ maps each state to the set of atomic propositions true in that state. A *path* in K is an infinite sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0$, $\langle w_i, w_{i+1}\rangle \in R$. We denote the suffix $w_i, w_{i+1}, \ldots$ of π by π^i. We define the size $\|K\|$ of K as $|W| + |R|$.

The notation $K, w \models \varphi$ indicates that a CTL* state formula φ holds at the state w of the Kripke structure K. Similarly, $K, \pi \models \psi$ indicates that a CTL* path formula ψ holds at a path π of the Kripke structure K. When K is clear from the context, we write $w \models \varphi$ and $\pi \models \psi$. Also, $K \models \varphi$ if and only if $K, w_0 \models \varphi$. The *model checking* problem is the following: given K and φ, is K a model for φ? It easy to see that the definitions of transition system and Kripke structure are equivalent [39].

The relation $\models$ is inductively defined as follows.

For all w, we have $w \models \mathbf{true}$ and $w \not\models \mathbf{false}$.

$w \models p$ for $p \in AP$ iff $p \in L(w)$.

$w \models \neg p$ for $p \in AP$ iff $p \notin L(w)$.

$w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$.

$w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$.

$w \models A\psi$ iff for every path $\pi = w_0, w_1, \ldots$, with $w_0 = w$, we have $\pi \models \psi$.

$w \models E\psi$ iff there exists a path $\pi = w_0, w_1, \ldots$, with $w_0 = w$, such that $\pi \models \psi$.

$\pi \models \varphi$ for a state formula φ, iff $w_0 \models \varphi$ where $\pi = w_0, w_1, \ldots$

$\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$.

$\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$.

$\pi \models X\psi$ iff $\pi 1 \models \psi$.

$\pi \models \psi_1 U \psi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \psi_2$ and for all $0 \leq j < i$, we have $\pi^j \models \psi_1$.

Containment and Simulation The problems that formalize correct trace-based and tree-based implementations of a system are *containment* and *simulation*, respectively. These problems are defined below with respect to two concurrent transition systems $P = \langle O, P_1, \ldots, P_n \rangle$ and $P' = \langle O', P'_1, \ldots, P'_m \rangle$ with $O \supseteq O'$, and with possibly different numbers of components. For technical convenience, we assume that $O = O'$.

The *containment problem* for P and P' is to determine whether $\mathcal{T}(P) \subseteq \mathcal{T}(P')$. That is, whether every trace accepted by P is also accepted by P'. If $\mathcal{T}(P) \subseteq \mathcal{T}(P')$, we say that P' *contains* P and we write $P \subseteq P'$.

While containment refers only to the set of computations of P and P', simulation refers also to the branching structure of the systems. Let c and c' be configurations of P and P', respectively. A relation $H \subseteq C \times C'$ is a *simulation relation* from $\langle P, c \rangle$ to $\langle P', c' \rangle$ iff the following conditions hold [87].

$H(c, c')$.

For all configurations $a \in C$ and $a' \in C'$ with $H(a, a')$, we have $L(a) = L(a')$.

For all configurations $a \in C$ and $a' \in C'$ with $H(a, a')$ and for every configuration $b \in C$ such that $succ_S(a, b)$, there exists a configuration $b' \in C'$ such that $succ_{S'}(a', b')$ and $H(b, b')$.

A simulation relation H is a *simulation from P to P'* iff for every $c \in C_0$ there exists $c' \in C'_0$ such that $H(c, c')$. If there exists a simulation from P to P', we say that P *simulates* P' and we write $S \preceq S'$. Intuitively, it means that the system P' has more behaviors than the system P. In fact, every tree embodied in P is also embodied in P'. The *simulation problem* is, given P and P', to determine whether $S \preceq S'$.

Image Computation The *image computation* problem is the following: given a set of states F, called the frontier, and a transition relation TR for it, calculate the image of F, that is the set of successors of each state in F according to TR.

OBDD: a data structure for transition systems representation Ordered Boolean decision diagrams (OBDDs) [30] are a canonical form representation for Boolean formulas. An OBDD is a rooted, directed acyclic graph with one or two terminal nodes labeled **0** or **1**, and a set of variable nodes of out-degree two. The variables respect a given linear order on all paths from the root to a leaf. Each path represents an assignment to each of the variables on the path. Since there can be exponentially more paths than vertices and edges, OBDDs can be substantially more compact than traditional representations like CNF. In many case, however, going from CNF representation to OBDD representation may cause an exponential blow-up [14].

2.1.4 An alternative definition of Transition Systems

In this section, we introduce the basic definitions about model checking that are needed in Chapter 4. We recall an alternative definition of transition system [85, 39] used by Model Checking tools; this definition is equivalent to the definition in Section 2.1 [39]. We use this alternative definition in Chapter 4 for the sake of the proof simplicity. We rephrase the Model Checking problem using this alternative terminology and definitions.

We follow the notation of [110, 109]. LTL (Linear Temporal Logic) is a modal logic aimed at encoding how states evolve over time. It has three unary modal operators (X, G, and F) and one binary modal operator (U). Their meaning is: $X\phi$ is true in particular state if and only if the formula ϕ is true in the next state; $G\phi$ is true if and only ϕ is true from now on; $F\phi$ is true if ϕ will become true at some time in the future; $\phi U\psi$ is true if ψ will eventually become true and ϕ stays true until then. We indicate with $L(O_1, \ldots, O_n)$ the LTL fragment in which the only temporal operators allowed are $O_1, \ldots, O_n$; for instance, $L(F, X)$ is the fragment of LTL in which only F and X are allowed.

The semantics of LTL is based on Kripke models. In the following, for an 'atomic proposition' we mean a Boolean variable. Given a set of atomic proposition, a Kripke structure for LTL is a tuple $\langle Q, R, \ell, I\rangle$, where Q is a set of states, R is a binary relation over states (the transition relation), ℓ is a function from states to atomic propositions (it labels every state with the atomic propositions that are true in that state), I is a set of initial states. A run of a Kripke structure is a Kripke model. A Kripke model for LTL is an infinite sequence of states, where the transition relation links each state with the one immediately following it in the sequence. The semantics of the modal operators is defined in the intuitive way: for example, $F\phi$ is true in a state of a Kripke model if ϕ is true in some following state.

The main problem of interest in practice is to verify whether all runs of a Kripke structure (all of its Kripke models) satisfy the formula; this is the Universal Model Checking problem. The Existential Model Checking one is to verify whether there is a run of the Kripke structure that satisfies the formula. In formal verification, we encode the behavior of a system as a Kripke structure, and the property we want to check as an LTL formula. Checking the structure against the formula tells whether the system satisfies the property. Since the Kripke structure is usually called a "model" (which is in fact very different from a Kripke model, which is only a possible run), this problem is called Model Checking.

In practice, all model checkers describe a system by the Kripke structure of its components. A Kripke structure can be seen as a transition system [39]. Thus the global system is obtained by parallel composition of the transition systems representing its components and sharing some variables [85, 39]; using this approach, we can give results that hold for all model checkers in Chapter 4.

Each component of the global system is modeled using a transition system, which is a formal way to describe a possible transition a system can go through. Intuitively, all is needed is to specify the state variables, the possible initial states, and which transitions are possible, i.e., we have to say whether the transition from state s to state s' is possible for any pair of states s and s'. The formal definition is as follows [85, 39].

Definition 8 *A finite-state transition system is a triple (V, I, ϱ), where $V = \{x_1, \ldots, x_n\}$ is a set of Boolean variables, I is a formula over V, and $\varrho(V, V')$ is a formula over $V \cup V'$, where $V' = \{x'_1, \ldots, x'_n\}$ is a set of new variables in one to one relation with elememts of V.*

Intuitively, V is the set of state variables, I is a formula that is true on a truth assignment if and only if it represents a possible initial state, and ϱ is true on a pair of truth assignments if they represent a possible transition of the system. The set of variables V' is needed because ϱ must refer

to both the value of a variable in the current state (x_i) and in the next state (x_i'). In other words, in this formula x_i means the value of x_i in the current state, while x_i' is the value of the same variable in the next state. For example, the fact that x_i remains true is encoded by $\varrho = x_i \rightarrow x_i'$: if x_i is true now, then x_i' is true, i.e., x_i is true in the next state.

Formally, a *state* s is an assignment to the variables; a state s' is *successor* of a state s iff $\langle s, s' \rangle \models \varrho(V, V')$. A *computation* is an infinite sequence of states $s_0, s_1, s_2, \ldots$, satisfying the following requirements:

Initiality: s_0 is initial, i.e. $s_0 \models I$

Consecution: For each $j \geq 0$, the state s_{j+1} is a successor of the state s_j

For the sake of simplicity, without loss of any generality, we only consider Boolean variables and Boolean assertions.

In order to model a complex system, we assume that each of its parts can be modeled by a transition system. Clearly, there is usually some interaction between the parts; as a result, some variables may be *shared* between the transition systems. In the following, we consider k transition systems $M_1, \ldots, M_k$. Every M_i is described by $((V_i^L \cup V_i^S), I_i(V_i), \varrho_i(V_i, V_i'))$ for i $1 \leq i \leq k$ where V_i^L is the set variables local to M_i, V_i^S is the set of shared variables of M_i, and $V_i = V_i^L \cup V_i^S$. A group of transition systems can be composed in different ways: synchronous, interleaved asynchronous, and asynchronous. The third way is not frequently used in Model Checking, so we only define the first two ways of composition. In the following, a process is any of the transition systems M_i.

The synchronous parallel composition of k transition systems is obtained by assuming that the global transition is due to all processes M_i making a transition simultaneously. In other words, all processes must make a transition at any time step, and no process is allowed to "idle" at any time step.

Definition 9 *The synchronous parallel composition of processes $M_1, \ldots, M_k$, is the transition system $M = (V, I, \varrho)$ described by:*

$$V = \bigcup_{i=1}^{k} V_i \qquad I(V) = \bigwedge_{i=1}^{k} I_i(V_i)$$

$$\varrho(V, V') = \bigwedge_{i=1}^{k} \varrho_i(V_i, V_i')$$

The synchronous parallel composition of $M_1, \ldots, M_k$, is denoted by $M_1 \| \ldots \| M_k$.

The basic idea of the interleaved asynchronous parallel composition is that only one process is active at the same time. As a result, a global transition can only result from the transition of a single process. The variables that are not changed by this process must maintain the same value.

Definition 10 *The interleaved asynchronous parallel composition of $M_1, \ldots, M_k$ is the transition system $M = (V, I, \varrho)$: , where V and I are as in the synchronous composition and ϱ is:*

$$\varrho(V, V') = \bigvee_{i=1}^{k} \left[\varrho_i(V_i, V_i') \wedge \bigwedge_{\substack{j=1 \\ j \neq i}}^{k} V_i^L = V_i^{L'} \right]$$

The interleaved asynchronous parallel composition of $M_1, \ldots, M_k$, is denoted by $M_1 | \ldots | M_k$.

A model can be described as the composition of transition systems. As a result, we can define the model checking problem for concurrent transition systems as the problem of verifying whether the model described by the composition of the transition systems satisfies the given formula.

2.1.5 Complexity and Compilability

Now, we introduce the definitions about complexity and compilability needed in Chapter 4.

We assume that the reader knows the basic concepts of complexity theory [111, 56]. What we mainly use in this chapter are the concepts of polynomial reduction and the class PSPACE.

The Model Checking problem is PSPACE-complete, and is thus intractable. On the other hand, as said in the Introduction, it makes sense to preprocess only one part of the problem (either the model or the formula), if this reduces the remaining running time. The analysis of how much can be gained by such preprocessing, however, cannot be done using the standard tools of the polynomial classes and reductions. The compilability classes [35] have to be used instead.

The way in which the complexity of the problem is identified in the theory of NP-completeness is that of giving a set of increasing classes of problems. If a problem is in a class C but is not in an inner class C′, then we can say that this problem is more complex to solve that a problem in C′. A similar characterization, with similar classes, can be given when preprocessing is allowed. For example the class $\|\!\rightsquigarrow$P is the class of problems that can be solved in polynomial time after a preprocessing step. Crucial to this definition are two points:

1. which part of the problem instance can be preprocessed?
2. how expensive is the preprocessing part allowed to be?

The first point depends on the specific problem and on the specific settings: depending on the scenario, for example, we can preprocess either the model or the formula for the model checking problem. The second question instead allows for a somehow more general answer. First, we cannot limit this phase to take polynomial time, as otherwise there would be no gain in doing preprocessing from the point of view of computational complexity. Second, we cannot allow the final result of this part to be exponentially large, for practical reasons; we bound the result of the preprocessing phase only to take a polynomial amount of space.

In order to denote problems in which only one part can be preprocessed, we assume that their instances are composed of two parts, and that the part that can be preprocessed is the first one. As a result, the model checking problem written as $\langle M, \phi \rangle$ indicates that M can be preprocessed; written as $\langle \phi, M \rangle$ indicates that ϕ can be preprocessed.

The "complexity when preprocessing is allowed" is established by characterizing how hard a problem is *after* the preprocessing step. This is done by building over the usual complexity classes: if C is a "regular" complexity class such as NP, then a problem is in the (non-uniform) compilability class $\|\!\rightsquigarrow$C if the problem is in C after a preprocessing step whose result takes polynomial space. In other words, $\|\!\rightsquigarrow$C is "almost" C, but preprocessing is allowed and will not be counted in the cost of solving the problem. More details can be found in [35].

In order to identify how hard a problem is, we also need a concept of hardness. Since the regular polynomial reductions are not appropriate when preprocessing is allowed, ad-hoc reductions (called nu-comp reductions in [35]) have been defined.

In this chapter, we do not show the hardness of problems directly, but rather use a sufficient condition called representative equivalence. For example, in order to prove that model checking is $\|\!\rightsquigarrow$PSPACE-hard, we first show a (regular) polynomial reduction from a PSPACE-hard problem to model checking and then show that this reduction satisfies the condition of representative equivalence.

Let us assume that we know that a given problem A is $\|\hookrightarrow$C-hard and we have a polynomial reduction from the problem A to the problem B. Can we use this reduction to prove the $\|\hookrightarrow$C-hardness of B ? Liberatore [83] shows sufficient conditions that should hold on A as well as on the reduction. If all these conditions are verified, then there is a nucomp reduction from $*A$ to B, where $*A = \{\langle x, y\rangle \mid y \in A\}$, thus proving the $\|\hookrightarrow$C-hardness of B.

Definition 11 (Classification Function) *A* classification function *for a problem A is a polynomial function $Class$ from instances of A to nonnegative integers, such that $Class(y) \leq ||y||$.*

Definition 12 (Representative Function) *A* representative function *for a problem A is a polynomial function $Repr$ from nonnegative integers to instances of A, such that $Class(Repr(n)) = n$, and that $||Repr(n)||$ is bounded by some polynomial in n.*

Definition 13 (Extension Function) *An* extension function *for a problem A is a polynomial function from instances of A and nonnegative integers to instances of A such that, for any y and $n \geq Class(y)$, the instance $y' = Exte(y, n)$ satisfies the following conditions:*

1. *$y \in A$ if and only if $y' \in A$;*
2. *$Class(y') = n$.*

Let us give some intuitions about these functions. Usually, an instance of a problem is composed of a set of objects combined in some way. For problems on boolean formulas, we have a set of variables combined to form a formula. For graph problems, we have a set of nodes, and the graph is indeed a set of edges, which are pairs of nodes. The classification function gives the number of objects in an instance. The representative function thus gives an instance with the given number of objects. This instance should be in some way "symmetric", in the sense that its elements should be interchangeable (this is because the representative function must be determined only from the number of objects). Possible results of the representative function can be the set of all clauses of three literals over a given alphabet, the complete graph over a set of nodes, the graph with no edges, etc. Let for example A be the problem of propositional satisfiability. We can take $Class(F)$ as the number of variables in the formula F, while $Repr(n)$ can be the set of all clauses of three literals over an alphabet of n variables. Finally, a possible extension function is obtained by adding tautological clauses to an instance. Note that these functions are related to the problem A only, and do not involve the specific problem B we want to prove hard, neither the specific reduction used. We now define a condition over the polytime reduction from A to B. Since B is a problem of pairs, we can define a reduction from A to B as a pair of polynomial functions $\langle r, h\rangle$ such that $x \in A$ if and only if $\langle r(x), h(x)\rangle \in B$.

Definition 14 (Representative Equivalence) *Given a problem A (having the above three functions), a problem of pairs B, and a polynomial reduction $\langle r, h\rangle$ from A to B, the condition of representative equivalence holds if, for any instance y of A, it holds:*

$$\langle r(y), h(y)\rangle \in B \quad \textit{iff} \quad \langle r(Repr(Class(y)), h(y)\rangle \in B$$

The condition of representative equivalence can be proved to imply that the problem B is $\|\hookrightarrow$C-hard, if A is C-hard [83]. As an example, we show these three functions for the $PLANSAT_1^*$ problem. $PLANSAT_1^*$ is the following problem of planning: giving a STRIPS [50] instance $y = \langle P, O, I, G\rangle$ in which the operators have an arbitrary number of preconditions and only one postcondition, is there a plan for y? $PLANSAT_1^*$ is PSPACE-Complete [32]. Without loss of generality we consider $y = (P, O \cup o_0, I, G)$, where o_0 is a operator which is always usable (it has no preconditions) and does nothing (it has no postconditions). We use the following notation:

$P = \{x_1, \ldots, x_n\}$, I is the set of conditions true in the initial state, $G = \langle \mathcal{M}, \mathcal{N} \rangle$. A state in STRIPS is a set of conditions. In the following we indicate with ϕ_i^h the hth positive precondition of the operator o_i, with ϕ_i all its the positive preconditions, with η_i^h its hth negative precondition, and with η_i all its negative preconditions; α_i is the positive postcondition of the operator o_i, β_i is the negative postcondition of the operator o_i. Since any operator has only one postcondition, for every operator i it holds that $\|\alpha_i \cup \beta_i\| = 1$.

Since we shall use them in the following, we define a classification function, a representative function and a extension function for $PLANSAT_1^*$:

Classification Function: $Class(y) = \|P\|$. Clearly it satisfies the condition $Class(y) \leq \|y\|$.

Representative Function: $Repr(n) = \langle P_n, \emptyset, \emptyset, \emptyset \rangle$, where $P_n = \{x_1, \ldots, x_n\}$. Clearly it is polynomial and satisfies the following conditions: (i) Class(Repr(n))=n, (ii) $\|Repr(n)\| \leq p(n)$ where p(n) is a polynomial.

Extension Function: Let $y = \langle P, O, I, G \rangle$ and $y' = Exte(y, n) = \langle P_n, O, I, G \rangle$. Clearly for any y and n s.t. $n \geq Class(y)$ y' satisfies the following conditions: (i)$y \in A$ iff $y' \in A$, (ii) $Class(y') = n$.

For the full definitions of compilability, the reader should refer to [35] for an introduction, to [34, 33] for an application to the succinctness of some formalisms, to [83] for further applications and technical advances.

2.2 Planning and Action Redundancy

In this section we introduce the basic definitions needed in Chapter 5, where we consider propositional STRIPS instances [50]. For the sake of simplicity, we neglect its many extensions. A STRIPS instance is a quadruple $\langle P, O, I, G \rangle$ where P is a set of conditions (a.k.a. facts, fluents, or variables), O is a set of operators (a.k.a. actions), I is the initial state, and G is the goal. A state is a subset of P: the state of the domain at some point is represented by the set of conditions that are true. An action is composed of four parts: the positive and negative preconditions and the positive and negative postconditions. These are simply the conditions that must be true or false to make the action executable, and the conditions that are made true or false by the execution of the action. The initial state is a state (the initial state is therefore fully known), while the goal is represented by a set of conditions that must be made true and a set that must be made false.

A plan is a sequence of actions that are executable in sequence from the initial state and lead to a state satisfying the goal. Redundancy is defined as follows.

Definition 15 (Redundant Action) *An action is redundant for a planning instance $\langle P, O, I, G \rangle$ iff there is a plan not containing it.*

In other words, a is a redundant action if and only if $\langle P, O \backslash \{a\}, I, G \rangle$ admits a plan that leads from the initial state to the goal.

2.3 Web Service and Process Algebra

Now, we introduce the concepts and the definitions needed for the Chapter 6.

2.3.1 LOTOS in a Nutshell

LOTOS is a specification language for distributed open systems normalized by the ISO [70]. It combines two specification models: one for static aspects (data and operations) which relies on the algebraic specification language ACT ONE [44] and one for dynamic aspects (processes) which draws its inspiration from the CCS [89] and CSP [66] process algebras (PA for short) .

Abstract Datatypes LOTOS allows the representation of data using algebraic abstract types. In ACT ONE, each *sort* (or datatype) defines a set of *operations* with arity and typing (the whole is called *signature*). A subset of these operations, the *constructors*, are sufficient to create all the elements of the sort. *Terms* are obtained from all the correct operation compositions. *Axioms* are first order logic formulas built on terms with variables; they define the meaning of each operation appearing in the signature.

Basic LOTOS This PA authorizes the description of dynamic behaviors evolving in parallel and synchronizing using rendez-vous (all the processes involved in the synchronization should be ready to evolve simultaneously along the same action). A process P denotes a succession of *actions* (also called event, channel or game in other formalisms) which are basic entities representing dynamic evolutions of processes; a process can be recursive. The symbol **stop** denotes an inactive behavior (it could be viewed as the end of a behavior) and the **exit** one depicts a normal termination. The specific **i** action corresponds to an internal (unobservable) evolution.

Now, we present LOTOS behavioral operators. The prefixing operator G**;**B proposes a rendez-vous on the action G, or an independent firing of this action, and then the behavior B is run. The non deterministic choice between two behaviors is represented using []. LOTOS has at its disposal three *parallel composition* operators. The general case is given by the expression B_1 |[G_1, ..., G_n]| B_2 expressing the parallel execution between behaviors $B1$ and $B2$. It means that $B1$ and $B2$ evolve independently except on the actions G_1, ..., G_n on which they evolve at the same time firing the same action (they also synchronize on the termination **exit**). Two other operators are particular cases of the former one to write out interleaving $B_1|||B_2$ which means an independent evolution of composed processes B_1 and B_2 (empty list of actions), and full synchronization $B_1||B_2$ where the composed processes synchronize on all actions (list containing all the actions used in each process). Moreover, the communication model proposes a multi-way synchronization: n processes may participate to the rendez-vous.

The disabling operator $B_1[>B_2$ model the interruption: the behavior B_1 could be interrupted at any moment by the behavior B_2; when B_1 is interrupted, B_2 is executed (without having interruptions).

Full LOTOS In this part, we describe the extension of basic LOTOS to manage data expressions, especially to allow value passing synchronizations. A process is parameterized by a (optional) list of formal actions $G_{i\in 1..m}$ and a (optional) list of formal parameters $X_{j\in 1..n}$ of type $T_{j\in 1..n}$. The full syntax of a process is the following:

$$\textbf{process } P\ [G_0, ..., G_m]\ (X_0{:}T_0, ..., X_n{:}T_n) : func := B\ \textbf{endproc}$$

where B is the behavior of the process P and $func$ corresponds to the functionality of the process: either the process loops endlessly (**noexit**), or it terminates (**exit**) possibly returning results of type $T_{j\in 1..n}$ (**exit**(T_0, ..., T_n)).

Action identifiers are possibly enhanced with a set of parameters (offers). An *offer* has either the form G**!**V and corresponds to the emission of a value V, or the form G**?**$X{:}S$ which means the reception of a value of type S in a variable X.

A behavior may depend on Boolean conditions. Thereby, it is possible that it be preceded by a guard [*Boolean expression*] $\rightarrow B$. The behavior B is executed only if the condition is true. Similarly, the guard can follow an action accompanied with a set of offers. In this case, it expresses that the synchronization is effective only if the Boolean expression is true (*e.g.*, G`?`X`:Nat[`X`>3]`). In the sequential composition operator, the left-hand side process can transmit some values (**exit**) to a process B (**accept**):

... **exit**(X_0, ..., X_n) $\gg$ **accept** Y_0:S_0, ..., Y_n:S_n **in** B

To end this section, let us say a word about CADP[3], a toolbox that supports developments based on LOTOS specifications. It proposes a wide panel of functionalities from interactive execution to formal verification techniques (minimization, bisimulation, proofs of temporal properties, compositional verification, etc).

2.3.2 Other Process Algebras

Numerous processes algebras have been proposed: CCS [89], CSP [66], ACP [12] are the basic ones. Extensions are π-calculus [96], Timed CSP [108]. Although syntactically different, all process algebras share a set of basic and dynamic constructs: actions, sequence, parallel composition, synchronizing actions, non deterministic choice, emission, reception, process, local process, recursive process.

2.3.3 Equivalences between processes

Two process are considered equivalent if their behavior is *indistinguishable* from an external observer interacting with them. In the process algebra community several notions of process equivalence have been proposed. More on the topic can be found in [89]. An approach is *trace-based*: two process are equivalent if they show the same execution traces. A process is contained in another one if the set of its execution traces are included in the set of execution traces of the other. Another approach is *tree-based*: two process are equivalent if they have equivalent execution trees, that is they simulate each other (they bisimulate). A process is simulated by another one if all its behaviors are contained in the behaviors of the other. A group of process running concurrently are simulated by another group of process running concurrently if all their behaviors are contained in the behaviors of the other. It is known that simulation implies containment. As example let us discuss Figure 2.2.

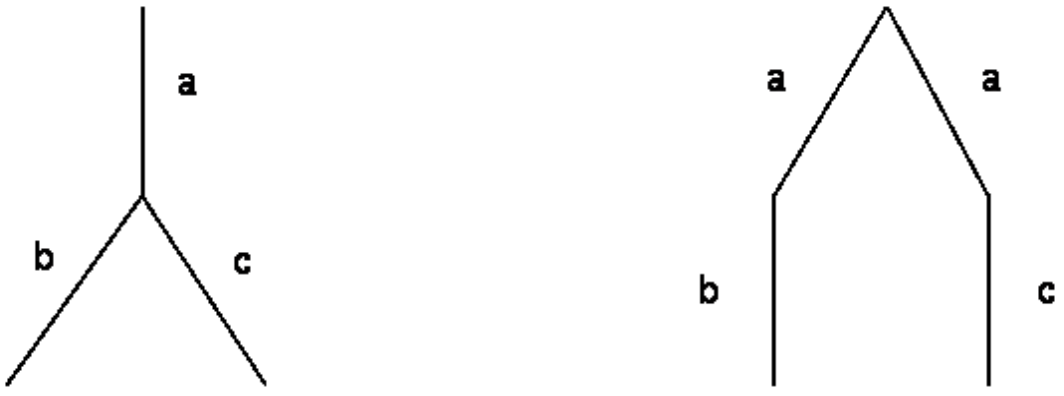

Figure 2.2: Processes Equivalences; a is a ticket purchase, b ticket use for the match, c a ticket change.

The left process corresponds in basic LOTOS to $a;(b[]c)$, the right one to $a;b\ []\ a;c$. They have the same traces (ab or ac), and so they are trace-equivalent. They do not bisimulate each other; after doing a the left process will do either b or c, while the right process on doing a, it will either

[3] http://www.inrialpes.fr/vasy/cadp/

choose to move in a state from which it does b or in a state from which it does$c$; depending on this choice, it cannot do one of the two actions whereas the left process leaves both possibilities open. Let a is a ticket purchase, b ticket use for the match, c a ticket change; the left process always allows to change the ticket after the purchase, the right one does not.

Chapter 3

Treewidth in Verification: Local *vs.* Global

3.1 Introduction

The *treewidth* of a graph measures how close the graph is to a tree (trees have treewidth 1). Many problems that are intractable (e.g. NP-hard, PSPACE-hard) for general graphs, are polynomial or linear-time solvable when the graph has bounded treewidth (see [27, 26, 25] for an overview). For example, constraint-satisfaction problems, which are NP-complete, are PTIME-solvable when the variable-relatedness graph has bounded treewidth [40, 54].

In [95, 61] the complexity of the model-checking problem is studied under the hypothesis of bounded treewidth; that is, it is assumed that the model is a state transition system, whose underlying graph has bounded treewidth. Bounding treewidth yields a large class of tractable model-checking problems. For example, while it is not know whether model checking μ-calculus formulas is in PTIME[71], it is in PTIME under the bounded treewidth assumption [95].

We refer to the treewidth of the state transition graphs of transition systems as the *global treewidth*. The global treewidth-boundedness assumption used in [61, 95] is not, in our opinion, useful to described real-world verification problems. There is little reason to believe, however, that the global treewidth of real-world systems is bounded. For example, it is easy to see that the graphs underlying systems with two counters are essentially grid, which are known to have high tree width [103]. In verification practice, real-world systems are often modeled as *concurrent* transition systems, where communication between concurrent components is modeled explicitly. When we consider the communication graph between the concurrent components (the component are the nodes, and an edge exists between each pair of communicating nodes), assuming treewidth boundedness is not unreasonable. Indeed, the topology of communication in concurrent systems is often constrained physically; for example, by the need to layout a circuit in silicon. Such topological constraints are studied, for example, in [86, 100]. In [86] the width of a Boolean circuit is related to the size of its corresponding OBDD, while in [100] bounded cutwidth is used to explain why ATPG, an NP-complete verification problem, is so easy in practice. Cutwidth boundedness is used also to improve symbolic simulation and Boolean satisfiability in [24, 116]. These various notions of bounded width are assumed because of the constrained topology of communication in concurrent systems.

In this chapter, we refer to treewidth of the component communication graph as *local* treewidth and study the impact of local-treewidth boundedness on the complexity of verification problems. We believe that because the component communication graph is often constrained physically, as noted above, assuming local treewidth boundedness is natural and realistic. (In fact, the assumption of treewidth boundedness is less severe than related assumption that are often made, such as *pathwidth*

boundedness or *cutwidth* boundedness [27, 26, 25].)

We first present a positive result. Under the hypothesis of global bounded treewidth, the complexity of the image computation in the explicit case decreases; the image computation in the explicit case is used in the algorithm [57], implemented in the model checker SPIN [67].

Another positive result can be found in [49], where it is proved that a CNF formula of bounded treewidth can be represented by an OBDD of polynomial size (treewidth here is defined on the primal graph of the formula, where vertices represent variables and edges represent the co-occurance of the variables in the same clause). Thus, if a transition relation of a concurrent transition system is specified by a CNF formula with bounded treewidth, then there is an OBDD [30] of polynomial size representing it. In contrast, the OBDD of transition relations often blow up, requiring symbolic model-checking techniques that avoid building these OBDDs [14]; this fact is formally proved in [48], and it is presented in Chapter 4 (Theorem 11). In [49] it is shown a first negative result about bounded local treewidth: the small-OBDD property of bounded treewidth CNF formulas is destroyed as soon as we apply existential quantification, which is a basic operation in symbolic model checking, where the image operations involves existential quantification [86].

In this chapter, we present another negative result: treewidth boundedness of a transition relation is not preserved under unrolling, which is a basic operation in SAT-based bounded model checking (BMC) [22]. (Note that while satisfiability of CNF formulas is NP-complete, satisfiability of bounded-treewidth CNF formulas can be solve in polynomial time, cf. [5]).

Finally, we show that the complexity of various verification problems are high even under the assumption of local treewidth boundedness. We review several verification problem for concurrent systems, including model checking, simulation, and containment, and show that the known lower bounds (PSPACE-complete, EXPTIME-complete, and EXPSPACE-complete, respectively [75, 64]) hold also under the assumption of local treewidth boundedness. (Our results are robust: the lower bound apply even under pathwidth boundedness or cutwidth boundedness.)

In summary, while global treewidth boundedness does have computational advantages, it is not a realistic assumption. In contrast, local treewidth boundedness is a realistic assumption, but its computational advantages are rather meager.

The chapter is organized as follows: in Section 3.2 we prove the result about the image computation, in section 3.3 we show the result about bounded model checking. Finally, in Section 3.4 we show that lower bound for model checking, simulation, and containment hold also under the assumption of local treewidth boundedness.

The results of this chapter are published in [49], and the basic definitions are introduced in Section 2.1.

3.2 Image Computation

In this section we give a result about the image computation operation in the explicit case; this is a key operation in all model checking algorithms that uses an explicit representation of the sequential transition system corresponding to the given concurrent transition system ([57], implemented in the model checker SPIN [67]).

Let P a sequential transition system, with $|V|$ states and let $|E|$ the cardinality of its transition relation. It is known that the worst case complexity of the image computation operation on P is $O(|V|^3)$, when P is represented explicitly (that is not using OBDD, or other symbolic representations).

If P has bounded treewidth, the image computation operation complexity in the explicit case decreases.

Theorem 1 *The worst case complexity of the image computation operation on a sequential transition system P with treewidth at most k is $O(|V|^2 \cdot k)$.*

Proof. It follows from Lemma 91 in [27], proved in [104]. This lemma states that if a graph $G = (V, E)$ has treewidth at most k, then $|E| \leq k \cdot |V| - k \cdot (k+1)/2$. □

3.3 Bounded Model Checking

Let us consider the effect of the local bounded treewidth on the complexity of Bounded Model Checking (BMC). In bounded model checking, given a transition relation $TR(V, V')$, k copies of the transition relation are created on $k+1$ copies of state variables where TR_1 uses V_0, V_1, TR_2 uses V_1, V_2, etc, in additional to initial and property constraints. In the following theorem, we show that BMC unrolling does not preserve the bounded treewidth.

Theorem 2 *Even though the transition relation of a concurrent transition system, represented by a CNF $TR(V, V')$, has bounded treewidth, its BMC unrolling can have unbounded treewidth.*

Proof. As an example, we take the case where the variables in $V = \{x_1, x_2, \ldots x_w\}$ is linearly arranged, and for the next state variable set V', each variable $x'_i = x_{i-1} \leftrightarrow x_i \leftrightarrow x_{i+1}$. The CNF for transition relation $TR(V, V')$ clearly has bounded pathwidth (where each path decomposition node consists of the variables $x_i, x_{i+1}, x'_i, x'_{i+1}$).

Now we considering Gaifman graph of the BMC unrolling. An example where two copies are unrolled is shown in Figure 3.1. The state variable for x_i at iteration j is denoted as x_i^j. We can see clearly that if we unroll, say, $h+2$ copies, the Gaifman graph will have a $w \times h$ grid as a minor, which implies unbounded pathwidth (and treewidth) [42].

□

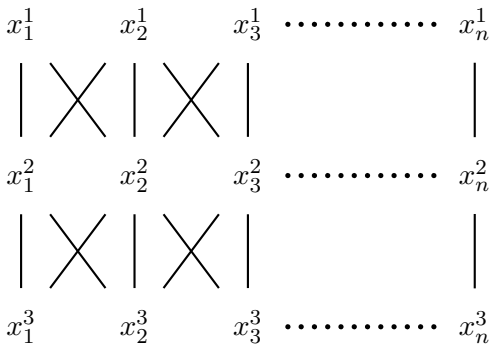

Figure 3.1: The $TR(k)$ in Theorem 2, for $k = 3$

3.4 Model Checking, Containment, Simulation

In this section we consider the complexity of the reachability, containment and simulation problems for concurrent transition systems, under the hypothesis of bounded treewidth both in the communication graph and in each component. The complexity of these problems has been studied in [75, 64]. We show that these problems have the same complexity of the general case, even if each component has constant size (and thus bounded treewidth and degree) and the communication graph has bounded pathwidth and degree (and then bounded cutwidth and treewidth). Our results are then robust; in fact a bounded pathwidth implies many other structural restrictions [107].

In [75] the model checking problem for temporal logics (e.g. CTL, LTL, CTL*) is shown to be PSPACE-hard, also in the reachability case. The reachability case is when the formula specifies an event that the transition system has to reach. For example in LTL, it is simply $F\psi$, where ψ is a Boolean formula. From the characteristic of the concurrent transition system used in the proof, the following theorem holds.

Theorem 3 *The CTL, LTL, and CTL* model checking for concurrent transition systems is* PSPACE*-hard also in the reachability case, and remains* PSPACE*-hard even if each component is fixed and the communication graph has bounded pathwidth and bounded degree.*

In [64] the simulation problem is shown to be EXPTIME-complete; from the characteristic of the concurrent transition systems used in the proof, the following theorem holds.

Theorem 4 *The simulation problem for concurrent transition systems is* EXPTIME*-hard, and remains* EXPTIME*-hard even if each component is fixed and the communication graph has bounded pathwidth and bounded degree.*

In [64] the containment problem is shown to be EXPSPACE-complete, but the concurrent transition systems used in the proofs have communication graphs with unbounded pathwidth and unbounded degree.

Theorem 5 *The containment problem for concurrent transition systems is* EXPSPACE*-hard, and remains* EXPSPACE*-hard even if each component has fixed size and the communication graph has bounded pathwidth and bounded degree.*

Proof. To prove hardness, we carry out a reduction from deterministic exponential-space-bounded Turing machines. Given a Turing machine T and input u of length n, we want to check whether T accepts the word u in space 2^n. we denote by Σ an alphabet for encoding runs of T (the alphabet Σ and the encoding are defined later). We write u' to represent the initial tape-encoding of u, i.e., if u is $u_1u_2\dots u_n$, u' is $(q_0,u_1)u_2\dots u_n$. We then construct a transition system P_T over the alphabet $\Sigma\cup\{\$\}$, for some $\$\notin\Sigma$, such that (i) the size of P_T is polynomial in $|T|$ and linear in n, and (ii) $\#u(\Sigma^\omega+(\Sigma^*\cdot\$^\omega))\subseteq\mathcal{T}(P_T)$ iff T does not accept the word u. The crucial point is that using bounded concurrency, we can handle the exponential size of the tape by n components that count to 2^n.

We assume, without loss of generality, that once T reaches a final state it loops there forever. The transition system P_T accepts all traces in Σ^ω, and accepts a trace $w\cdot\$^\omega\in\Sigma^*\cdot\$^\omega$ if either

1. w is not an encoding of a prefix of a legal computation of T,
2. w is an encoding of a prefix of a legal computation of T, but, within this prefix, the computation still has not reached a final state, or
3. w is an encoding of a prefix of a legal, but rejecting, computation of T over any input.

Thus, P_T rejects a trace $w\cdot\$^\omega$ iff w encodes a prefix of a legal accepting computation of T and the computation has already reached a final state. Hence, P_T accepts all traces in $\#u(\Sigma^\omega+\Sigma^*\cdot\$^\omega)$ iff T does not accept the word u.

Now to the details of the construction. Let $T=\langle\Gamma,Q,\mapsto,q_0,F_{acc},F_{rej}\rangle$, where Γ is the alphabet, Q is the set of states, and $\mapsto\colon(Q\times\Gamma)\to(Q\times\Gamma\times\{L,R\})$ is the transition function. We write $(q,a)\mapsto(q',b,\delta)$ for $\mapsto(q,a)=(q',b,\delta)$, with the meaning that when in state q and reading a in the current tape cell, T moves to state q', writes b in the current tape cell and moves its head one cell to the left or right, depending on δ. Finally, q_0 is T's initial state, $F_{acc}\subseteq Q$ is the set of final accepting states, and $F_{rej}\subseteq Q$ is the set of final rejecting states.

We encode a configuration of T by a string in $\#\Gamma^*(Q\times\Gamma)\Gamma^*$, of the form $\#\gamma_1\gamma_2\dots(q,\gamma_i)\dots\gamma_{2^n})$. The meaning of this is that the j'th cell, for $1\le j\le 2^n$, is labeled γ_j, T is in state q and its head points to the i'th cell.

We encode a computation of T by a sequence of configurations, which is a word over $\Sigma=\{\#\}\cup\Gamma\cup(Q\times\Gamma)$. Let $\#\sigma_1\dots\sigma_{2^n}\#\sigma'_1\dots\sigma'_{2^n}$ be two successive configurations of T in such a sequence. (Here, each σ_i is in Σ.) If we set $\sigma_0=\sigma_{2^n+1}=\#$ and consider a triple $\langle\sigma_{i-1},\sigma_i,\sigma_{i+1}\rangle$, for $1\le i\le 2^n$, it is clear that the transition function of T prescribes σ'_i. In addition, along the encoding of the entire computation, # must repeat exactly every 2^n+1 letters. Let $next(\sigma_{i-1},\sigma_i,\sigma_{i+1})$ denote our expectation for σ'_i. That is, with the γ's denoting elements of Γ, we have:

$next(\gamma_{i-1}, \gamma_i, \gamma_{i+1}) = next(\#, \gamma_i, \gamma_{i+1}) = next(\gamma_{i-1}, \gamma_i, \#) = \gamma_i$.

$next((q, \gamma_{i-1}), \gamma_i, \gamma_{i+1}) = next((q, \gamma_{i-1}), \gamma_i, \#) = \begin{cases} \gamma_i & \text{if } (q, \gamma_{i-1}) \mapsto (q', \gamma'_{i-1}, L) \\ (q', \gamma_i) & \text{if } (q, \gamma_{i-1}) \mapsto (q', \gamma'_{i-1}, R) \end{cases}$

$next(\gamma_{i-1}, (q, \gamma_i), \gamma_{i+1}) = next(\#, (q, \gamma_i), \gamma_{i+1}) = next(\gamma_{i-1}, (q, \gamma_i), \#) = \gamma'_i$,
where $(q, \gamma_i) \mapsto (q', \gamma'_i, \delta)$. [1]

$next(\gamma_{i-1}, \gamma_i, (q, \gamma_{i+1})) = next(\#, \gamma_i, (q, \gamma_{i+1})) = \begin{cases} \gamma_i & \text{if } (q, \gamma_{i+1}) \mapsto (q', \gamma'_{i+1}, R) \\ (q', \gamma_i) & \text{if } (q, \gamma_{i+1}) \mapsto (q', \gamma'_{i+1}, L) \end{cases}$

$next(\sigma_{2^n}, \#, \sigma'_1) = \#$.

A necessary and sufficient condition for a trace to encode a legal computation of T on the word u is that consecutive configurations are compatible with *next*.

Now for the construction of P_T. P_T is a concurrent process with $n+1$ components. The first component, P_M is the master process that accept all the traces Σ^ω, and accept all non-accepting traces in $\Sigma^* \cdot \$^\omega$. The other components $P_1, \cdots, P_n$, are used by P_M and their only task is perform the count as in Example 1; each of these processes is associated with a bit (P_1 with the least significant, P_n the most significant).

Let us describe the process P_M. In spirit, P_M follows the outline of the master process in [64]. In the construction of P_M, we use the following block of states G_{Σ^3}, which is used to generate sequences of triples $(\sigma_{i-1}, \sigma_i, \sigma_{i+1}) \in \Sigma^3$. G_{Σ^3} have $|\Sigma^3|$ states, each representing a triple, and labeled by the middle state. For two triples (u, u', u'') and (v, v', v''), there is an transition from the first to the second iff $u' = v$ and $u'' = v'$. P_M can either start in a clique of Σ states to generate Σ^ω, or it can start in a block of states (which we call *Init*) to generate non-accepting traces. All edges in *Init* have condition *true*. From a state s in *Init*, we can reach a corresponding successor state, which represents the same triple as the successors of s in *Init*, in a new block of states B_s, of which every state asserts c_0 to start the count in the component P_1. In other words, $c_0 = \bigvee_{t \in B_s | s \in Init} t$. All edges into states in B_s have condition *true*, except those that go into states with label $next(s)$. As P_M progresses in B_s, the counter is counting to 2^n. The edges into the state labeled with $next(s)$ have condition $\neg s^n_{10}$, and from every state in B_s, we can move to a state which is a self loop labeled \$ with condition s^n_{10}. This asserts that the trace we are generating is not a prefix of a legal computation over T. Alternatively, P_M can also start in a clique of size $|\Sigma'|$ where $\Sigma' = \{\#\} \cup \Gamma \cup \{(Q - F_{acc}) \times \Gamma\}$, i.e., all the non-accepting symbols in Σ. Each edge in the clique have condition *true*, and each state in the clique can go to the self loop on \$ on condition *true*. This captures all the (legal or illegal) non-accepting traces on T.

It is easy to see that $|P_T|$ is polynomial in $|T|$ and linear in n. The processes $P_M, P_1, \cdots, P_n$ have constant size. P_M interacts only with P_1 and with P_n, the generic P_i interacts only with P_{i-1} and P_{i+1}: the communication graph is a ring and then it has bounded pathwidth and degree.

Now, given the word $u = u_1 u_2 \dots u_n$, we construct P to be a concurrent transition system that generates the language $\#(q_0, u_1)u_2 \dots u_n(\Sigma^\omega + (\Sigma^* \cdot \$^\omega))$. In fact, P can be easily taken to be a concurrent transition system with $n+1$ components, each with $|\Sigma| + 1$ states, implemented as a shifter. In other words, the next state of component i is the current state of component $i+1$, and component $n+1$ can non-deterministically generate $\Sigma^\omega + (\Sigma^* \cdot \$^\omega)$. Obviously, each component is of constant size, and the concurrent transition system is of bounded pathwidth and bounded degree. It follows that T does not accept the word u iff $P \subseteq P_T$. By taking T to be an universal Turing machine, we showed that the containment problem for concurrent transition systems is EXPSPACE-hard even if each component has fixed size and the communication graph has bounded pathwidth and bounded degree.

□

[1] We assume that T's head does not "fall" from the right or the left boundaries of the tape. Thus, the case where $i = 1$ and $(q, \gamma_i) \mapsto (q', \gamma'_i, L)$ and the dual case where $i = 2^n$ and $(q, \gamma_i) \mapsto (q', \gamma'_i, R)$ are not possible.

Chapter 4

Model Checking, Preprocessing, and BDD Size

4.1 Introduction

Temporal Logic Model Checking [39] is a verification method for discrete systems. In a nutshell, the system, often called the model, is described by the possible transitions of its components, while the properties to verify are encoded in a temporal modal logic. It is used, for example, for the verification of protocols and hardware circuits [14, 59]. Many tools, called *model checkers*, have been developed to this aim. The most famous ones are SPIN [67] and SMV [86] (with its many incarnations: NuSMV [37], RuleBase [15]), VIS [29], and FormalCheck [65].

There are many languages to express the model; the most widespread ones are Promela and SMV. Two temporal logics [101] are mainly used to define the specification: CTL [39] and LTL [99]. In this chapter we focus on the latter.

In many cases, the two inputs of the model checking problem (the model and the formula) can be processed in a different way. If we want to verify several properties of the same system, it makes sense to spend more time on the model alone, if the verification of the properties becomes faster. Many tools allow to build the model separately from checking the formula [36, 114, 69]; in this way, one can reuse the same model, compiled into a data structure, in order to check several formulae.

In the same way, we may wish to verify the same property on different systems: the property is this time the part we can spend more time on. Many tools allow populating a property database [36, 114, 69], i.e., a collection of temporal formulae which will be checked on the models. We imagine a situation in which we early establish the requirements that our system must satisfy, even before the system is actually designed. As a result, and we can fill a database of temporal formulae, but we do not yet describe the system. While the design/modeling of the system goes on, we can preprocess the formulae (without knowledge of the model, which is not yet known). Whenever the system is specified, we can then use the result of this preprocessing step to check the model against the formulae.

In this chapter, we analyze whether preprocessing a part of the model checking problem instances improve the performances. The technical tool we use is [35, 83] the compilability theory. This theory characterizes the complexity of problems when the problem instances can be divided into two parts (the fixed and the varying part), and we can spend more time on the first part alone, provided that the result of this preprocessing step has polynomial size respect the fixed part. We show that the Model Checking problem remains PSPACE-hard even if we can preprocess either the model or the formula, if this preprocessing step is constrained to have a polynomial size. These theorems hold for all model checkers.

Finally, we answer to a long-time standing question in Symbolic Model Checking. It has been

observed that the BDDs that are used by SMV and other Symbolic Model Checking systems become exponentially large in some cases. However, it has not yet been established whether this size increase is due to the choice of variable ordering, or to the kind of BDDs employed, or it is intrinsic of the problem. We show that, if PSPACE $\not\subseteq \Pi_2^p \cap \Sigma_2^p$, such a grown is, in the worst case, unavoidable. This result is independent from the particular class of BDDs and from the variable order of the BDDs. It also holds for all decision diagrams representing integer-value functions whose evaluation problem is in the polynomial hierarchy, such as BEDs [117], RBCs [3], MTBDDs [38], and ADDs [13].

The results of this chapter are published in [48], and the basic definitions are introduced in Sections 2.1.4 and 2.1.5.

4.2 Results

The Model Checking problem for concurrent transition systems is PSPACE-complete [75]. In Section 4.2.1, we prove that the following problems remain PSPACE-hard even if preprocessing is allowed (in other words, they are $\|\not\rightarrow$PSPACE-hard):

1. model checking on the synchronous and interleaved asynchronous composition of transition systems, where the transitions systems are the fixed part of the problem and the LTL formula is the varying part;

2. the same problem, where the LTL formula is the fixed part and the transition system is the varying part;

3. given a set of transition systems and a formula as the fixed part, a state as the varying part, checking whether the state is a legal initial state.

We can conclude that preprocessing the model or the formula does not lead to a polynomial algorithm for model checking. We recall that the fixed part is preprocessed off-line in a polynomial data structure during the preprocessing phase, and the varying part is given on-line.

The relevance of the first two problems is clear: in formal verification, it is often the case that many properties (formulae) have to be verified over the same system (the model, in this case modeled by the transition systems); on the other hand, it may also be that the same property has to be verified on different systems.

The result about the third problem is less interesting by itself. On the other hand, we use it to prove that the superpolynomial growth of the size of the data structures (e.g. OBDDs) currently used in model checkers based on the Symbolic Model Checking algorithms [86] (such as SMV and NuSMV) cannot be avoided in general. The result is independent from its variable ordering, and it holds for others data structures that can be employed. We show these results in Section 4.2.2.

We point out that most of Temporal Logic Model Checking algorithms [39] fall in one of three classes: Symbolic Model Checking algorithms, which work on symbolic representation of M; algorithms based on Bounded Model Checking [23] (i.e. based on reduction from Model Checking into SAT); algorithms that work on an explicit representation of M (e.g. [57]). Our results concerning the size of the BDD (or some other decision diagrams) are valid for all algorithms of the first class.

In the proofs of the following sections we consider Existential Model Checking problems, but the results are valid also for the Universal case; in fact PSPACE is closed under complementation also for compilability.

4.2.1 Preprocessing Model Checking

We now identify the complexity of the Model Checking problem when the preprocessing of the model (represented as the composition of transition systems) is allowed, both in the synchronous and in the interleaved case.

Theorem 6 *The model checking problem for k synchronous concurrent process* $MC_{syn} = \langle (M_1 || \dots || M_k), \varphi \rangle$ *where* $\varphi \in LTL$ *is* $\|\!\rightsquigarrow$PSPACE*-hard, and remains* $\|\!\rightsquigarrow$PSPACE*-hard for* $\varphi \in L(F, G, X)$.

Proof. It is similar to the proof of the Theorem 7. We carry out a reduction from the $PLANSAT_1^*$ problem, that satisfies the conditions of representative equivalence; the main difference is about the LTL formula. □

We now consider the Model Checking problem for concurrent processes composed in a interleaved way when the model can be preprocessed.

Theorem 7 *The model checking problem for k interleaved concurrent process* $MC_{asyn} = \langle (M_1 | \dots | M_k), \varphi \rangle$ *where* $\varphi \in LTL$ *is* $\|\!\rightsquigarrow$PSPACE*-complete, and remains* $\|\!\rightsquigarrow$PSPACE*-hard for* $\varphi \in L(F, G, X)$.

Proof. We show a reduction, that translates an instance $y \in PLANSAT_1^*$ into an instance $\langle r(y), h(y) \rangle \in M_{asyn}$, satisfying the condition of representative equivalence. Given $y = \langle P, O, I, G \rangle \in PLANSAT_1^*$

- $r(y)$ defines a concurrent transition systems $M_1, \dots, M_n$, where each M_i is obtained from a variable $x_i \in P$ and it is described by:

  $V_i = \{x_i\}$
  $I_i(V_i) = (x_i) \vee (\neg x_i)$
  $\varrho_i(V_i, V_i') = (x_i = 0 \wedge x_i' = 0) \vee (x_i = 0 \wedge x_i' = 1) \vee$
  $(x_i = 1 \wedge x_i' = 0) \vee (x_i = 1 \wedge x_i' = 1)$

  The process $M = M_1 \| \dots \| M_n$ represents all possible computations, starting from all possible initial assignments, over the variables $x_1, \dots, x_n$.

- $h(y) = h(I, G, O) = \neg(\phi_I \wedge \phi_G \wedge \phi_O)$
  where:

  $$\varphi_I = \bigwedge_{i \in I} x_i \wedge \bigwedge_{i \notin I} \neg x_i$$
  $$\varphi_G = F\Big(\bigwedge_{i \in \mathcal{M}} x_i \wedge \bigwedge_{i \in \mathcal{N}} \neg x_i \Big)$$
  $$\varphi_O = G \bigvee_{i=0}^{m} \Big[\bigwedge_{h=1}^{\|\phi_i\|} \phi_i^h \wedge \bigwedge_{h=1}^{\|\eta_i\|} \neg \eta_i^h \wedge X\gamma_i \wedge \bigwedge_{\substack{j \neq i \\ j=1}}^{n} (x_j \leftrightarrow X x_j) \Big]$$

  where

  $$\gamma_i = \begin{cases} \alpha_i & \text{if} \quad \alpha_i \neq \emptyset \\ \neg \beta_i & \text{if} \quad \beta_i \neq \emptyset \end{cases}$$

φ_I adds constraints about the initial states of y represented by I.

φ_G adds constraints about the goal states of y represented by G: it tells that a goal state will be reached.

φ_O describes the operators in O: globally (i.e. in every state) one of the operators must be used to go in the next state; φ_O also describes the nop operator o_0.

Now, we prove that $y \in PLANSAT_1^*$ iff $\langle r(y), h(y)\rangle \in M_{asyn}$. Given $y = \langle P, O, I, G\rangle$, a solution for y is a plan which generates the following sequence of states: $(s_1, \ldots, s_p)$ where s_1 is an initial state and s_p is a goal state. This sequence of states is obtained applying a sequence of operators $(o_{h_1}, \ldots, o_{h_p})$ chosen in $O = \{o_1, \ldots, o_m\}$ in the following way: for all i s.t. $1 \leq i \leq p$, preconditions for o_{h_i} are included in the state s_i, and the state s_{i+1} is obtained from the state s_i modifying the postcondition associated with o_{h_i}. We remark that a state in STRIPS is the set of conditions.

The model $M = r(y) = r(P)$ represents all possible traces starting from all possible initial configurations, over the variables $x_1, \ldots, x_n$. Thus, in this case the Existential Model Checking problem $\langle M, \varphi\rangle$ reduces to the satisfiability problem for φ: we check whether ther exists a trace among all traces over the variables $x_1, \ldots, x_n$ that satisfies the LTL formula φ. Therefore, we have to prove that $y \in A$ iff $\varphi = h(y)$ is satisfiable:

$\Rightarrow$. Given a solution for $y \in A$, we identify a model for $\varphi = h(y)$; by construction such a model has:

- initial state s_1^M s.t. $\ell(s_1) = I \cup \{\neg x_i | x_i \notin I\}$
- a state s_p^M s.t. $\ell(s_p) \subseteq \mathcal{M} \cup \{\neg x_i | x_i \notin \mathcal{N}\}$
- given a state s_i^M, s_{i+1}^M is successor of s_i^M iff
  - $\ell(s_i^M) \subseteq Precond(o_{h_i})$, where $Precond(o_{h_i}) = \{x_j | x_j \in \phi_{h_i}\} \cup \{\neg x_j | x_j \in \eta_{h_i}\}$
  - $\ell(s_{i+1}^M) = \ell(s_i^M) \cup \alpha_i - \beta_i$
    where α_i is the positive postcondition of o_{h_i} and β_i is the negative postcondition of o_{h_i}.
- an infinite number of states: when the state s_p is reached this state is repeated for at least once or for ever (applying the nop operator o_0), or it is possible, it depends from y, to apply any operators whose preconditions are satisfied by $\ell(s_p^M)$.

$\Leftarrow$. Let $(s_1^M, \ldots, s_p^M, \ldots)$ a model for φ, and let s_p the goal state, that the first state satisfying φ_G. We obtain the sequence of states visited by a plan which is a solution for y, by cutting the states after the goal state s_p and assigning $s_i = \ell(s_i^M)$; thus this sequence of states $(s_1, \ldots, s_p)$, associated with the plan, has by construction:

- initial state s_1 s.t. $s_1 = I \cup \{\neg x_i | x_i \notin I\}$
- a state s_p s.t. $s_p \subseteq \mathcal{M} \cup \{\neg x_i | x_i \notin \mathcal{N}\}$
- given a state s_i, s_{i+1} is successor of s_i iff
  - $s_i \subseteq Precond(o_{h_i})$
  - $s_{i+1} = s_i \cup \alpha_i - \beta_i$
    where α_i is the positive postcondition of o_{h_i} and β_i is the negative postcondition of o_{h_i}.

□

Now we show the complexity results, both in the synchronous and in the interleaved case, when the formula can be preprocessed.

Theorem 8 *The model checking problem for k synchronous concurrent process* $MC'_{syn} = \langle \varphi, (M_1 || \dots || M_k) \rangle$ *where* $\varphi \in LTL$ *is* $\|\!\rightarrowtail$PSPACE*-complete, and remains* $\|\!\rightarrowtail$PSPACE*-hard for* $\varphi \in L(F, G, X)$.

Proof. $PLANSAT_1^*$ is the following problem of planning: giving a STRIPS [50] instance $y = \langle P, O, I, G \rangle$ in which the operators have an arbitrary number of preconditions and only one postcondition, is there a plan for y? $PLANSAT_1^*$ is PSPACE-complete [32]. Without loss of generality we consider $y = (P, O \cup o_0, I, G)$, where o_0 is a operator which is always usable (it has no preconditions) and does nothing (it has no postconditions). We use the following notation: $P = \{x_1, \dots, x_n\}$, I is the set of conditions true in the initial state, $G = \langle \mathcal{M}, \mathcal{N} \rangle$. A state in STRIPS is a set of conditions.

In the following we indicate with ϕ_i^h the hth positive precondition of the operator o_i, and with η_i^h the hth negative precondition of the operator o_i; α_i is the positive postcondition of the operator o_i, β_i is the negative postcondition of the operator o_i. Since any operator has only one postcondition, for every operator i it hold that $\|\alpha_i \cup \beta_i\| = 1$.

We show a polynomial reduction from the problem A to the problem B that satisfies the condition of representative equivalence. This proves that B is $\|\!\rightarrowtail$C-hard, if A is C-hard; to apply this condition we must define a *Classification Function*, a *Representative Function* and a *Extension Function* for A. Thus we use such a proof schema: we define a *Classification Function*, a *Representative Function* and a *Extension Function* for $PLANSAT_1^*$, then we show a polynomial reduction from an instance $y \in PLANSAT_1^*$ to an instance $\langle r(y), h(y) \rangle \in MC'_{SYN}$ that satisfies the condition of representative equivalence.

Let $y = \langle P, O, I, G \rangle \in PLANSAT_1^*$. We define r and h as follows:

- $r(y) = r(P) = \neg \left\{ F(x_g) \wedge G \bigwedge_{i=0}^{n} \left[\neg(x_i \leftrightarrow X x_i) \rightarrow \bigwedge_{\substack{j=1 \\ j \neq i}}^{n} (x_j \leftrightarrow X x_j) \right] \right\}$
- $h(y)$ defines the transition systems $M_1 \| \dots \| M_k$. The generic M_i is obtained from the operators $o_{i_1}, \dots, o_{i_{d_i}}$ whose postcondition involves the variable $x_i \in P$; d_i is the number of such operators. We add the variable x_g; thus we have at most as many processes as variables: if k is the number of variables used as postcondition of operators plus one, we have $k \leq n + 1$. Let M_k the process associated with the variable x_g; this variable is 0 at the beginning and it becomes 1 only when the goal of the $PLANSAT$ problem is reached. M_i, for i s.t. $1 \leq i < k$, is defined by:

$$V_i = \bigcup_{q=1}^{d_i} \phi_{i_q} \cup \eta_{i_q} \cup \alpha_{i_q} \cup \beta_{i_q}$$

$$I_i(V_i) = \bigwedge_{x_j \in I \cap V_i} x_j \wedge \bigwedge_{x_j \in \overline{I} \cup V_i} \neg x_j$$

$$\varrho_i(V_i, V_i') = \bigvee_{k=1}^{d_i} \bigwedge_{h=1}^{\|\phi_{i_k}\|} \phi_{i_k}^h \wedge \bigwedge_{h=1}^{\|\eta_{i_k}\|} \neg \eta_{i_k}^h \wedge \neg \Big(\bigwedge_{i \in \mathcal{M}} x_i \wedge \bigwedge_{i \in \mathcal{N}} \neg x_i \Big) \wedge (x_i' \equiv b_{i_k})$$

$$\text{where } b_{i_k} = \begin{cases} 1 & \text{if } \alpha_{i_k} \neq \emptyset \\ 0 & \text{if } \beta_{i_k} \neq \emptyset \end{cases}$$

The process M_k is defined by:

$$V_k = \{x_g\}$$

$$I_k(V) = (x_g = 0)$$

$$\varrho_k(V_k, V_k') = \textstyle\bigwedge_{i \in \mathcal{M}} x_i \wedge \bigwedge_{i \in \mathcal{N}} \neg x_i \wedge x_g' = 1$$

Now we prove that this reduction is correct, i.e. $y \in PLANSAT_1^*$ iff $\langle r(y), h(y)\rangle \in MC'_{SYN}$.

$\Rightarrow$. Given a solution for $y \in PLANSAT_1^*$, we show a path of M which satisfies φ (r(y) defined above).

A solution for y is a plan which generates the following sequence of states: $(s_1, \ldots, s_p)$ where s_1 is a initial state and s_p is a goal state. This sequence of states is obtained by applying a sequence of operators $(o_{h_1}, \ldots, o_{h_p})$.

By construction M admits a path $(s_1^M, \ldots, s_p^M, s_{p+1}^M, \ldots)$ s.t.:

- $\ell(s_i^M) = s_i \cup \neg x_g$ for i $1 \leq i \leq p$
- $\ell(s_{p+1}^M) = s_p \cup x_g$

This path satisfies φ:

- φ does not constrain about the initial state, therefore every initial state of the model is legal;
- $x_g \subseteq \ell(s_{p+1}^M)$, therefore $F(x_g)$ is true;
- the path shown is s.t. only one variable change at a time, therefore the subformula under the Globally is true.

$\Leftarrow$. Given a path of M which satisfies φ, we show a solution for $y \in PLANSAT_1^*$.
The path is a sequence $(s_1^M, \ldots, s_p^M, s_{p+1}^M, \ldots)$. We can obtain the sequence of states visited by a plan for y in this way:

- $s_i = \ell(s_i^M) - \{\neg x_g\}$ for i $1 \leq i \leq p$;
- we ignore the rest of the path of M.

□

Theorem 9 *The model checking problem for k interleaved concurrent process* $MC'_{asyn} = \langle \varphi, (M_1|\ldots|M_k)\rangle$ *where* $\varphi \in LTL$ *is* $\|\!\rightsquigarrow$PSPACE*-complete, and remains* $\|\!\rightsquigarrow$PSPACE*-hard for* $\varphi \in L(F)$.

Proof. We carry out a reduction from the $PLANSAT_1^*$ problem, that satisfies the conditions of representative equivalence. The proof is similar to the proof of the Theorem 8. □

Now we introduce the decision problem $MC_{s_0} = \langle [M, \varphi], s_0\rangle$, where M is specified by the interleaved parallel composition of k transition systems $M_1, \ldots, M_k$, $\varphi \in L(F)$, and s_0 is a specific state. MC_{s_0} is true if the model checking problem for concurrent transition system $\langle M, \varphi\rangle$ has solution and s_0 is a legal initial state i.e., is an initial state belonging to M that satisfies φ.

Theorem 10 MC_{s_0} *is* $\|\!\rightsquigarrow$PSPACE*-complete.*

Proof. The hardness follows from a polynomial time reduction from the problem $\langle (P, O, G), I\rangle$, that can be easily shown $\|\!\rightsquigarrow$PSPACE-complete on the basis of the results in [84].

We sketch the reduction. We encode each operator in O into each process M_i, and the goal G into the formula φ. We encode the set of initial states I using s_0. □

4.2.2 The Size of BDDs

In this section we prove that the size of BDDs and others data structures increases superpolynomially with the size of the input data, in the worst case, when are used in a Symbolic Model Checking algorithm.

Let M a model specified by k concurrent transition systems $M_1, \ldots, M_k$, and let φ an LTL (or a CTL or CTL*) formula.

Theorem 11 *If* PSPACE $\not\subseteq \Pi_2^p \cap \Sigma_2^p$*, then there is not always a BDD of any kind and with any variable order that is polynomially large and represents the set of initial states consistent with* M *and* φ*.*

Proof. The evaluation problem for any kind of BDD, i.e. giving a BDD and an assignment of its variables evaluate the BDD, is in P . If there exists a poly-size BDD representing the set of initial states consistent with M and φ, then we can compile M and φ in the BDD and evaluate the assignment (representing a initial state) in polynomial time. This implies that MC_{s_0} is in $\|\!\rightarrow$P. We know from Theorem 10 that MC_{s_0} is $\|\!\rightarrow$PSPACE-complete. Therefore if such a BDD exists, then $\|\!\rightarrow$PSPACE=$\|\!\rightarrow$P. Now, by applying Theorem 2.12 in [35], we conclude that there is no poly-size reduction from MC_{s_0} to the evaluation problem for a BDD, if PSPACE $\not\subseteq \Pi_2^p \cap \Sigma_2^p$. □

Symbolic Model Checking algorithms work by building a representation of the set of the initial states of M that satisfy φ. In particular, this set is represented by BDDs. Therefore, the last theorem proves that these algorithms, in the worst case, end up with a BDD of superpolynomial size. This result does not depend on the kind of BDD used (free, ordered, etc.) and on the variable ordering. On the contrary, it holds also when the states are labeled with enumerative variable; in other words it holds not only for BDD but also for any decision diagram, provided that the evaluation problem over this representation of the states is in a class of the polynomial hierarchy. More formally, we consider an arbitrary representation of a set of states. The evaluation problem is that of determining whether a state belongs to a set.

Theorem 12 *Given a method for representing a set of states whose evaluation problem is in a class* Σ_i^p *of the polynomial hierarchy, it is not always possible to represent in polynomial space the set of legal initial states of a model* M *and a formula* φ*, provided that* $\Sigma_{i+1}^p \neq \Pi_{i+1}^p$*.*

The proof of this theorem has the same structure of the proof of the Theorem 11.

Instances of such data structures, currently used in Symbolic Model Checking tools, are BDDs, Boolean Expression Diagrams (BEDs) [117] and Reduced Boolean Circuits (RBCs) [3]. Our results hold also for data structures used to represent integer-value functions, like Multi terminal binary decision diagrams (MTBDDs) [38], Algebraic Decision Diagrams (ADDs) [13]; see for details the survey [43].

Now, we discuss related works. There are several results on the exponential growth of the BDD size respect to a particular problem (e.g. integer multiplication [31]); moreover there are results dealing with the size growth of other decision diagrams [43] respect to particular problems. While these results are not conditional to the collapse of the polynomial hierarchy as the ones reported in this chapter, they are also more specific, as they concern only specific kinds of data structures (e.g. OBDDs) respect to particular problems (e.g. integer multiplication). On the other hand, it is also possible to prove that the above two theorems cannot be stated unconditionally: indeed if P = PSPACE, then there is a data structure of polynomial size allowing the representation of the set of initial states in such a way deciding whether a state is in this set can be decided in polynomial time. As a result, the non-conditioned version of the above two theorems implies a separation in the polynomial hierarchy.

Chapter 5

The Complexity of Checking Action Redundancy

5.1 Introduction

In this chapter, we discuss some problems related to the reachability property, that is interesting also in the field of reasoning about actions. In particular, we consider the planning area, where the reachability problem is equivalent to the existence of a combination of actions that reaches a given goal. The connection between Model Checking and Planning are discussed and used in [58, 18, 20, 77, 98, 21, 19].

Most problems in reasoning about action and in planning, like plan existence and plan generation, are problems on a fixed planning domain: the initial states, goal, and possible actions are assumed fixed and cannot be modified. This is not always the case, in fact, in many real-world applications the domain can be (at least to some extent) modified. In this chapter, we study a number of problems concerning a modifications of the set of available actions from the domain. For example, in an industrial production setting, the actions might correspond to physical machinery that performs an action. In this setting, deciding whether an action (machinery) is necessary to reach the goal makes sense.

It is important to remark that we are not checking whether an action can be removed from a single plan [51, 52, 74], a problem that arises naturally in the context of plan abstraction [118, 6]. Rather, we are checking whether an action can be removed from the set of available actions. Such a problem makes sense in several situations such as:

Design: if the planning instance is the formalization of a system that is yet to be built, it makes sense to consider whether some actions can be removed, as this may correspond to the simplification of the design;

Reliability: in some system, operation must be warranted regardless of faults; since the effect of faults on a planning instance is to make some actions non-executable, then all actions should be redundant to ensure that the system will work properly in all cases;

Solving: the cost of solving a planning instance is often related to the number of possible actions; knowing that a specific action is not really needed may simplify the planning generation problem (this is the motivation behind the work of Nebel, Dimopolous, and Koehler [93].)

An application domain of our problems are Web services. Web services (WSs) are distributed and independent pieces of code solving specific tasks which communicate with each other through

the exchange of messages. A more unusual specificity that distinguishes them from more traditional software components is that they are deployed and then accessed through the internet. Planning techniques can be applied in the Web service synthesis and composition: For example, in [113] a new (not existing) WS is specified by the goal, and the existing WSs (the components) are described by the planning actions. The resulting plan represent the interaction between the components and the new WS. In this framework our approach can be useful:

> we can characterize the set of the existing WSs that need to be reliable. This has consequences in the quality of service requirements: if a service is not redundant, then the provider of the service has to ensure an higher level of reliability.
>
> we can establish how much longer becomes the shortest plan (that is the interaction) if we remove a service.
>
> if we reduce the services needed to compose a new service, we more easily understand the behaviour of the synthesized service (it is more human readable).

It is important to note that solving the problem of action redundancy is done *before* the plan is generated. We are not considering the problem of removing an action from a plan after that the plan has been found. This topic has already been studied [51, 52]; rather, we are considering the problem of the possible removal of an action before any plan is generated. In other words, the problem is not to establish the redundancy of an action in a plan, but the redundancy of an action in a planning domain. There are scenarios in which the problem of redundancy in a plan makes sense, and others in which redundancy in a domain is more relevant.

Various problems are considered. First, we consider problems related to actions only: given a set of actions, is there any action that can be simulated by the other ones? In other words, is there any action that is redundant? In this case, we are not (yet) considering a specific initial state nor a specific goal. This question is therefore of interest whenever either we do not have yet an initial state or goal, or we want to study the problem for all possible initial states or goals. We call these problems Absolute Action Redundancy.

We also consider some problems about planning domains in which the initial states and goal are fixed. Actually, **the most relevant case is that in which the initial states and the goals are only partially known; however, the complexity results for the case of full knowledge carry on to the case of partial knowledge of the initial state and the goal.** The problems considered for this case are: is a specific action redundant (Single Specific Action Redundancy)? is there any action that can be removed (Single Action Redundancy)? and problems related to finding a minimal set of actions.

In this chapter, we study the computational complexity of these problem. An assumption we make is that all actions have the same cost, so that minimality is considered in terms of the number of actions. Another implicit assumption is that all actions are independent, in the sense that it is possible to remove a single one of them from the domain. These assumptions are not always realistic: for example, it may be that two actions are both executed by the same part of a system: the relevant problem is whether this part is necessary, which is equivalent to the redundancy of both actions. These extensions of the problems considered in this chapter are under investigation.

The chapter is organized as follows. In the next section, we introduce the notion of redundancy and define the problems we study. We assume that notions of computational complexity and planning are known. In the third section we give some preliminary results that will be used in the complexity analysis. In the fourth section we show the complexity of problems related to redundancy. We conclude the chapter by comparing our results with related work presented in the literature. The results of this chapter are published in [47], and the basic definitions are introduced in Section 2.2.

We now describe the computational problems we consider. In the following, we assume that the initial state and the goal are either fully specified or not specified at all. Clearly, the most interesting problems are those in which these two parts of the domain are only partially specified. The reason for not considering this case is simply that all hardness proofs extend from the fully specified to the partially specified case, and that the membership proofs are easy to extend because non-deterministic polynomial space is equal to polynomial space. In other words, the restriction to the fully specified case is done only for the sake of simplicity, but all results carry on the more interesting case of partial specification.

We first focus on the redundancy of actions in a given planning instance.

Single Specific Action Redundancy. Given a planning instance $\langle P, O, I, G\rangle$ and an action $a \in O$, is a redundant in $\langle P, O, I, G\rangle$?

Single Action Redundancy. Given a planning instance $\langle P, O, I, G\rangle$, is there a redundant action in O?

The latter problem is similar to the former one, but we are not checking whether a specific action is redundant but whether there is a redundant action in O.

While the two above problems are about the redundancy of actions for a given initial state and goal, we also consider the redundancy of actions when neither the initial state nor the goal are specified.

Absolute Specific Action Redundancy

Given: $\langle P, O\rangle$ and an action $a \in O$

Question: is a redundant? In other words, is it true that $\langle P, O, I, G\rangle$ has plans if and only if $\langle P, O\backslash\{a\}, I, G\rangle$ has plans for every I and G?

Absolute Action Redundancy

Given: $\langle P, O\rangle$

Question: is there any redundant action in O?

The following problems are related to minimizing the number of actions.

Minimal Number of Actions

Given: a planning instance $\langle P, O, I, G\rangle$ and an integer k with $k < |O|$

Question: do $O_k \subset O$, where $|O_k| = k$, exist s.t. $\langle P, O_k, I, G\rangle$ has plans?

Minimal Set of Actions

Given: a planning instance $\langle P, O, I, G\rangle$ and $O' \subset O$

Question: is O' minimal? (minimal = does not contain any redundant action)

Specific Action of a Minimal Set

Given: a planning instance $y = \langle P, O, I, G\rangle$ and $a \in O$

Question: is a in a minimal subset of $O' \subset O$ such that $\langle P, O', I, G\rangle$ has plans?

Finally, we consider four problems related to the length of plans.

Plan Length for a Specific Action Subset

Given: a planning instance $y = \langle P, O, I, G\rangle$ and $O' \subset O$

Question: does $y' = \langle P, O', I, G\rangle$ have a plan of the same length of the shortest plans for y?

Plan Length for an Action Subset

Given: a planning instance $y = \langle P, O, I, G\rangle$

Question: does $O' \subset O$ exist s.t. $y' = \langle P, O', I, G\rangle$ has a plan of the same length of the shortest plans for y?

Plan Length Increase for a Specific Action Subset

Given: a planning instance $y = \langle P, O, I, G\rangle$, $O' \subset O$, and an integer $c > 0$

Question: does $y' = \langle P, O', I, G\rangle$ have a plan of length $l + c$ where l is the length of the shortest plans for y?

Plan Length Increase for an Action Subset

Given: a planning instance $y = \langle P, O, I, G\rangle$ and an integer $c > 0$

Question: does $O' \subset O$ exist s.t. $y' = \langle P, O', I, G\rangle$ has a plan of length $l + c$ where l is the length of the shortest plans for y?

5.2 Useful Properties

Here we present two properties that will be useful in the sequel. The first such property states that two planning instances can be combined in such a way the plans of the resulting instance are related to the plans of the two original ones.

Formally, we are given two planning instances $\langle P_1, O_1, I_1, G_1\rangle$ and $\langle P_2, O_2, I_2, G_2\rangle$, which we assume built on disjoint sets of conditions and operators. The disjunction of these two planning instances is defined as follows.

Definition 16 *The disjunction of $\langle P_1, O_1, I_1, G_1\rangle$ and $\langle P_2, O_2, I_2, G_2\rangle$ is $\langle P, O, I, G\rangle$, where:*

$$\begin{aligned} P &= P_1 \cup P_2 \cup \{x\} \\ O &= O_1 \cup O_2 \cup \{o_1, o_2\} \\ I &= I_1 \cup I_2 \\ G &= \langle \{x\}, \emptyset\rangle \end{aligned}$$

where $\{x\}$ is a new condition and $\{o_1, o_2\}$ are new operators; o_1 has G_1 as precondition and x as a positive postcondition, while o_2 has G_2 as precondition and x as a positive postcondition.

The plans of $\langle P, O, I, G\rangle$ are the plans of $\langle P_1, O_1, I_1, G_1\rangle$ with o_1 added at the end, the plans of $\langle P_2, O_2, I_2, G_2\rangle$ with o_2 at the end, plus any other sequence that is obtained by interleaving other actions to these plans. In a way, the new planning instance is a "disjunction" of the original instances, as it has all plans of both. The minimal plans of the disjunction are exactly the shortest of the plans of the two instances.

The second property states that a planning instance can be modified in such a way that the resulting instance always has plans composed of all operators, in addition to the plans of the original one.

Definition 17 *The maximized versions of a* STRIPS *instance* $\langle P, O, I, G\rangle$ *is the instance* $\langle P \cup Y \cup Z \cup \{w\}, B \cup C \cup \{d\}, I, G'\rangle$*, where* Y *and* Z *are sets of new variables of the same cardinality of* O*,* w *is a new variable,* B *and* C *are sets of new actions of the same cardinality of* O*, and* d *is a new action. The goal* G' *is similar to* G*, but it also requires* w *to be false. The effects of the actions are:* c_i *is the same as* a_i*, but also makes* y_i *true;* b_i *has no precondition, but makes* z_i*,* w*, and the preconditions of* c_i *true. Finally,* d *is only applicable if all variables* $Y \cup Z$ *are true and makes the goal satisfied.*

This instance has plans: the sequence $[b_1, c_1, \ldots, b_m, c_m]$ makes all variables $Y \cup Z$ true; the application of d therefore makes the goal reached. In order for this plan to work, no action can be removed from it: removing d leads to having w true in the final state; on the other hand, d is only applicable if all variables $Y \cup Z$ are true, which can be accomplished only if all actions $B \cup C$ have been applied at least once.

On the other hand, if the original instance is satisfiable, then the plans of the original instance can be mapped into plans of this new one just be replacing each a_i with the corresponding c_i. No other sequence of actions is a plan: if a sequence contains any of the b_i, then it has to contain d as well, as b_i makes w true and d is the only action that makes it false, as required by the goal. In turn, d is applicable only if all variables in $Y \cup Z$ are true, which means that all actions $B \cup C$ must been applied beforehand.

5.3 Complexity of Action Redundancy

In this section, we characterize the complexity of the problems we have introduced. Most of these problems belong to PSPACE, as they can be solved by solving a number of regular planning problems, such as plan existence, that are known to be in PSPACE. The difficult part of their complexity characterization, indeed, is the hardness part.

The first problem we consider is the Single Specific Action Redundancy. The redundancy of a single specific action has been called *complete irrelevance* by Nebel, Dimopolous, and Koehler [93], who proved the following theorem.

Theorem 13 ([93]) *Single Specific Action Redundancy is* PSPACE*-complete.*

A related question is: given a planning instance for which we *know* a plan, does the removal of an action a cause the domain not to have plans any more? This problem is still PSPACE-complete.

Using the above result we can easily show that:

Theorem 14 *Single Action Redundancy is* PSPACE*-complete.*

Proof. Membership. This is the problem of checking whether a planning instance contains an action that can be removed. In other words, it is equivalent to solve the Single Specific Action Redundancy for all actions. Since solving each of these problems is in PSPACE, the problem of Single Action Redundancy is in PSPACE as well.

Hardness. Proved by reduction from the problem of plan existence. Given a STRIPS instance for which we want to establish the plan existence, we build its maximized version. This maximized version is an instance that has the same plans of the original instance plus some plans composed of

all actions. If the original instance does not have plans, the maximized version has only the plans composed of all actions; as a result, no action is redundant. If the original instance has some plans, even if they contains all actions, the maximized version has the same plans, which however do not contain the action d because this action is not present in the original instance. As a result, the modified instance contains a redundant action if and only if the original instance has plans. □

Let us now consider the problems of absolute redundancy. These are the problems of checking redundancy of actions when neither the initial state nor the goal are known. As for the case of (non-absolute) redundancy, we consider first the redundancy of a single specific action and then the presence of a redundant action.

Theorem 15 *Absolute Specific Action Redundancy is* PSPACE-*complete.*

Proof. Membership. We are given a set of actions O, and want to check whether an action $a \in O$ is not necessary for whatever initial state and goal. In other words, for any possible I and G, we have to check whether the existence of plans $\langle P, O, I, G\rangle$ implies the existence of plans in $\langle P, O \backslash \{a\}, I, G\rangle$. The problem can be therefore solved by nondeterministically guessing I and G and then solving a problem in PSPACE. The problem is therefore in PSPACE.

Hardness. We show a polynomial reduction from the problem of checking the existence of a plan for the instance $\langle P, O, I, G\rangle$. We build the Absolute Specific Action Redundancy instance $\langle P, O \cup a\rangle$ where a is a new action that is only executable in the state I and has postcondition G. Clearly, this action a is redundant if and only if the goal can be reached from the initial state. □

Theorem 16 *Absolute Action Redundancy is* PSPACE-*Complete.*

Proof. Membership. We solve the Absolute Specific Action Redundancy problem for each of the actions in O. Since we solve a polynomial number of problems that can all be solved in polynomial space, the problem is in PSPACE.

Hardness. We show a polynomial reduction from Absolute Specific Action Redundancy. Let the instance of this problem be $\langle P, O\rangle$ where $O = \{a_1, .., a_n, a\}$ and a is the "candidate" redundant action. The idea of the reduction is to make all actions but a irredundant. To this aim, we consider an action a'_i for each a_i: this action has the same preconditions and postconditions of a_i plus a new postcondition y_i. The instance we build is $\langle P \cup \{y_1, .., y_n\}, O'\rangle$ where $O' = \{a'_1, .., a'_n, a\}$. None of the actions a'_i is redundant, as a'_i is the only action that makes y_i true. As a result, a is redundant for $\langle P, O\rangle$ iff $\langle P \cup \{y_1, .., y_n\}, O'\rangle$ contains a redundant action. □

Let us now consider the problems related to minimality. The first one is that of checking the existence of a group of k actions that are sufficient for building a plan. It is easy to see that this problem is not the same as checking the existence of a plan of length k, as an action may occur more than once in a plan. In other words, we are checking the plans containing a minimal number of actions, not the plans of minimal length.

Theorem 17 *Minimal Number of Actions is* PSPACE-*Complete.*

Proof. Membership. Guess a subset of $O' \subseteq O$ composed of k actions, and check the existence of plans for the instance in which there are only actions O'. Since the problem can be solved by a nondeterministic guess followed by the solution of a PSPACE problem, the problem is in NPSPACE and is therefore in PSPACE.

Hardness. We show a polynomial reduction from Single Action Redundancy. The instance $\langle P, O, I, G\rangle$ contains a redundant action if and only if there exists a set of $k = |O| - 1$ actions that are sufficient to make the goal reachable. □

Checking whether a set of actions is minimal is the complementary problem of Single Action Redundancy: a set of actions is minimal if and only if it does not contain any redundant action.

Corollary 1 *Minimal Set of Actions is* PSPACE*-complete.*

We now consider the problem of checking whether an action is in some minimal set of actions.

Theorem 18 *Specific Action of a Minimal Set is* PSPACE*-complete.*

Proof. Membership. For each $O' \subseteq O$, we check whether this set of actions is sufficient for the existence of a plan; if it is, we check it for minimality; if it is minimal, we check whether $a \in O'$. This algorithm requires only a polynomial amount of space.

Hardness. We show a reduction from the problem of plan existence. Given a STRIPS instance, we build its maximized version, and check whether d is in some minimal set of actions. The plans of the maximized version are composed of all actions plus any plan of the original instance. As a result, if the original instance has no plans, then no action is redundant. On the other hand, if the original instance has plans, then the goal is reachable even if the action d is removed. As a result, d is not in any minimal set of actions. □

The problem of redundancy considered so far are about whether actions can be removed from a domain while preserving the plan existence. We now consider the problem of whether the removal of actions increases the length of the minimal plans.

Theorem 19 *Plan Length for a Specific Action Subset is* PSPACE*-Complete.*

Proof. Membership. Find the minimal plan length for both the original instance and the instance without the actions to remove. Both problems can be solved in polynomial space.

Hardness. Given an instance $\langle P, O, I, G\rangle$ with n conditions, we build the composition of this instance with a planning instance that is known to have plans longer than 2^n. The resulting instance has plans that are made of plans of $\langle P, O, I, G\rangle$ followed by the action o_1 plus plans of the exponential-plan instance followed by o_2 (plus other plans that are obtained by interleaving these plans with other actions.)

We can now exploit the fact that $\langle P, O, I, G\rangle$ has plans if and only if it has plans of length bounded by 2^n. As a result, the minimal plans of the composed instances are those of $\langle P, O, I, G\rangle$ followed by o_1 if this instance is satisfiable, or the plans of the second instance followed by o_2 otherwise. As a result, removing o_1 leads to an increase of the minimal plan length if and only if the instance $\langle P, O, I, G\rangle$ has plans. □

The following theorem is about the similar problem in which there is no specific set of actions to remove: we are only checking whether some actions can be removed without increasing the minimal plan length.

Theorem 20 *Plan Length for an Action Subset is* PSPACE*-Complete.*

Proof. Membership. Check the minimal plan length for the original instance and for all instances obtained from it by removing some actions. All these problems can be solved in polynomial space.

Hardness. Proved by reduction from plan existence. We modify the planning instance in such a way a plan composed of all actions always exists. We then create the translation of this instance on a new alphabet, with the only exception that d is replaced by two actions a and b, which have the same effect when executed in sequence. Finally, we compose the two planning instances, making sure that the actions o_1 and o_2 are duplicated.

This instance is built in such a way that every action can be removed without increasing the plan length. There is however an exception: if all minimal plans contain d, then the removal of d can only be compensated by the two actions a and b, which means that the minimal plan length is increased. In turns, this is only possible if the original instance has no plan. □

The problems with the similar formulation but where the minimal plan length is allowed to increase of a given number k are clearly as hard as the ones above, where $k = 0$. The membership of these problems to PSPACE can be proved by modifying the proofs for the case $k = 0$.

Corollary 2 *Plan Length Increase for a Specific Action Subset and Plan Length Increase for a Action Subset are* PSPACE-*Complete.*

Finally, we note that all hardness results about a specific instance $\langle P, O, I, G\rangle$ extend to the case where this instance is not fully specified. More precisely, if a problem about instances $\langle P, O, I, G\rangle$ is extended to the case where only a part of I and G is given, then hardness results maintain their validity. For example, the problem of a specific action redundancy, which consists in checking whether an action $a \in O$ can be removed from $\langle P, O, I, G\rangle$, can be extended to the case of partial knowledge of I and G by assuming that some of the conditions are not (yet) known in the initial state, and some conditions are not known whether they will be part of the goal. This is exactly what we expect when the domain is fixed but the initial state and goal are not completely known. In this case, an action is redundant if it so for any possible initial state and goal. Since the case of full specification is simply a particular case, the hardness proof holds for the general case as well. Moreover, the general problem can be formalized as: for any I and G that extends the known parts of the initial state and goal, solve the problem of action redundancy for the case of full specification. Since the latter problem is in PSPACE, cycling over all possible I and G does not increase complexity, and the problem therefore remains in PSPACE. In summary, **the complexity of all problems that are stated with a fixed I and G extends to the case of partially specified initial state and goal.**

5.4 Related Work

The problem of checking the redundancy of actions in a domain is clearly related to the problem of checking the existence of a plan, and to the problem of finding such a plan. These are the two most studied problems in planning, and a large number of papers on this topic are in the literature. Our results clearly build on the work of Bylander [32], Nebel [93, 92, 11], and Bäckström [7, 9, 10, 8].

The paper which presents results strictly related to our work is the paper by Nebel, Dimopoulos, and Koehler [93] where they analyze the problem of checking the redundancy of STRIPS instances. More precisely, the problem they considered is whether a specific action or a specific fact (condition) is really needed to achieve the goal. They also considered the problem of whether an action or a fact is needed by *some* of the plans of the domain. Their paper is mainly focused on finding irrelevant actions of facts to the aim of simplifying the plan search. Most of their work is therefore devoted to developing heuristics, and only one complexity result is given (checking whether a fact or action is redundant in a STRIPS instance is PSPACE-complete). The present work extends Nebel, Dimopoulos, and Koehler's by giving a more complete complexity characterization of redundancy in planning.

A concept related to checking redundancy of an action in a domain is that of redundancy of an action in a specific plan. Removing redundant actions in a plan is a problem that has been investigated in the context of planning by abstraction [118, 74, 6], where a plan is called *justified* if it does not contain actions that can be removed. This idea has been formalized in a number of different ways by Fink and Yang [51, 52]. Given the similarity with the work reported in the present chapter, it is important to clarify the differences:

1. in the work on justified planning, a plan is given, and the goal is to remove actions from it; on the other hand, in action redundancy, the work has to be done on the domain, not on a specific plan;

2. a plan is justified if it does not contain actions that can be removed; on the other hand, it may be that a plan contains two times the same action, and only one of them can be removed; in other words, justification is not the irredunancy of actions, but rather the irredundancy of *action occurrences* in a plan; on the contrary, if an action is redundant then it can be removed altogether from the domain.

Clearly, whether justification or redundancy is of interest depends on the stage of planning: if the system on which planning is needed can be modified, then action redundancy is of interest; once the domain is fixed, and a plan has been found, the aim of reducing the plan length is of interest.

The problem of checking the redundancy of actions in a specific plan is also of interest in the context of plan recognition. It is indeed clear that not all actions may be directed toward a single goal; for example, it has been observed that "operators of control system, when not otherwise occupied, often glance at the current value of process variables" [60]. These 'checking' actions are often redundant to the plan, and not taking this fact into account may produce a wrong plan identification.

The problems studied in this chapter are about planning domains where actions can be removed. A similar assumption that has been considered by several authors [73, 63, 94, 82] is that the initial state and the goal may change. The aim of plan modification or plan adaptation is to change an already known plan to take into account the different initial state or goal.

In spite of a name homonymy, redundancy in partially ordered planning search spaces [72] is not really related to redundancy as intended in this chapter. Roughly speaking, redundancy in a search space is the presence of solutions that are not really necessary to visit. Clearly, this kind of redundancy is about plans, and is about searching for plans, while action redundancy is about a domain and is not directly related to a particular planning algorithm.

5.5 Conclusions and Future Work

While the results of this chapter are about the redundancy of actions in a domain, several other problems can be considered if we remove the assumption that the planning domain cannot be modified. For example, costs can be associated to actions, and the aim of the redundancy elimination would be to reduce the overall cost of the actions. If actions consume resources, one can consider whether we can reduce the cost associated to an action (for example, we may replace a part of a system with a more efficient one).

A restriction we have (implicitly) assumed in this chapter is that actions can be eliminated one-by-one. This is however not always true, as the removal of an action is only effective if all actions that require a part of a system are removed. In such cases, we are no longer interested in removing a single action, but rather into removing groups of actions. These extensions are currently under investigation.

In our opinion, it is also interesting to study the complexity of these problems in the non deterministic case, with different levels of observability (total, partial, none). In this chapter we have shown the complexity in the deterministic case with full observability.

Finally, we are studying efficient algorithms to solve these problems in order to apply these techniques to the Web service synthesis problem.

Chapter 6

Web Services: a Process Algebra Approach

6.1 Introduction

In this chapter, we present an application of formal verification to Web services. Our aim is to reduce the effort of testing phase in the WS development.

Web services (WSs) are distributed and independent pieces of code solving specific tasks which communicate with each other through the exchange of messages. A more unusual specificity that distinguishes them from more traditional software components is that they are deployed and then accessed through the internet. Some XML-based standardized technologies have already been proposed to support WSs development: WSDL interfaces abstractly describe messages to be exchanged, SOAP is a protocol for exchanging structured information, UDDI is used to publish and discover WSs, BPEL4WS (BPEL for short) is a notation for describing executable business process behaviors. WSs raise many theoretical and practical issues which are part of on-going research. Some well-known problems related to WSs are to specify them in an adequate, formally defined and expressive enough language, to compose them (automatically), to discover them through the web, to ensure their correctness.

Formal methods provide an adequate framework (many specification languages and reasoning tools) to address most of these issues (description, composition, correctness). Different proposals have emerged recently to abstractly describe WSs, most of which are grounded on transition system models (Labelled Transition Systems, Mealy automata, Petri nets, etc.) [16, 68, 91, 62, 80, 55].

With respect to these works, we use process algebras (PAs for short) as abstract representation. Process algebras offer more respect to all these previous approaches: they not only provide temporal logic model checking, but also bisimulation (resp. simulation) analysis, that is we can establish whether two processes have equivalent behaviors (resp. whether one of the two includes the behavior of the other). Bisimulation analysis is useful to establish when a service can substitute another services in a composition [28]; another use of bisimulation is to check the redundancy of service in a community.
Because process algebras support simulation analysis, we can apply to WSs a well-know design method, the *hierarchical refinement* [78, 76]: intuitively we start with an abstract description of a process and we refine it iteratively, obtaining at each step a less abstract one. At each stage, using simulation and bisimulation we can verify the correspondence between the current version and the previous (more abstract) one. It can be applied also in the BPEL modelling of WSs, using the two-way mapping. Moreover we argue, with a simple consideration, that the simulation can be part of the problem of automatic composition of services.

In Figure 6.1 we present a framework, for the design and verification of WSs using process

algebras [17] (*e.g.* CCS, π-calculus, LOTOS). In this chapter we focus on LOTOS, one of the most expressive process algebra. We provide a two-way mapping between BPEL/WSDL and LOTOS, and general guidelines for translations between BPEL/WSDL and a process algebra. We choose LOTOS because it allows us the data handling, and the verification and the modelling of the BPEL handlers.

Respect to the quoted previous works, we study also the direction from a formal language to BPEL. Using the two-way mapping, that allows an automatic translation between the two languages, two choices are available: designing in BPEL and verifying with a process algebra, designing and verifying in a process algebra. These two approaches are not alternative, but they can be combined in the same development.

Designing in BPEL and verifying with a process algebra. Going from BPEL to a PA allows us the verification step in PA, and the converse allows to see the counterexamples directly in BPEL, hopefully even in the visual interface for designing BPEL services. Obviously one can correct in PA, and the BPEL corrected code is automatically generated. This approach is useful also for reverse engineering issues, and when we want to verify BPEL services developed by others.

Designing and verifying in a process algebra. We point out that using the mapping we can automatically obtain BPEL/WSDL specifications. To our knowledge this is the first work in this direction. As advocated in a previous work [105], being simple, abstract and formally defined, PAs make it easier to specify the message exchange between WSs, and to reason on the specified systems. They are especially worthy as a first description step because they enable one to analyze the problem at hand, to clarify some points, to sketch a (first) solution using an abstract language (then dealing only with essential concerns), to have at one disposal a formal description of one or more services-to-be, therefore adequate to use existing reasoning tools to verify and ensure some temporal properties (safety, liveness and fairness properties), behavior equivalences (bisimulation), and execution traces. Process algebras design allows the distributed development and software reuse.

In Section 6.2 we focus on the two-way mapping between LOTOS to BPEL and we give the guidelines formalizing the translation between process algebras and BPEL. In Section 6.3 we illustrate the features provided by our approach: temporal logic model checking, execution traces, simulation, bisimulation. We discuss the hierarchical refinement and other problems that we can solve using a process algebra representation for WSs and a bisimulation analysis. In Section 6.4 presents related works and motivates our contribution with respect to them. We draw up concluding remarks in Section 6.5 and we mention some future works.

The results of this chapter are published in [46, 106], and the basic definitions are introduced in Section 2.3.

6.2 The two-way mapping between LOTOS and BPEL

In this section we show the two-way mapping between LOTOS, a process algebra that allows data handling, and BPEL. Our goal is showing a two-way mapping between the two languages, that allows an automated translation. For lack of space, it is not possible to introduce all of BPEL, XMLSchema, and XPath. Accordingly, the reader who is not used with them should refer to [4, 1, 2].

When it is possible, we present together both directions of the mapping. While the translation from BPEL to LOTOS implicitly preserves the BPEL structure, the converse does not: LOTOS allows to use the construct in very flexible manner, BPEL does not. In the LOTOS design we have to be careful, if we want a simple automatic translation, to write behavior structurally similar to BPEL ones. For example in BPEL a service can communicate only with other services, there is no message exchange inside a service. In LOTOS instead, as in all process algebras, there are no

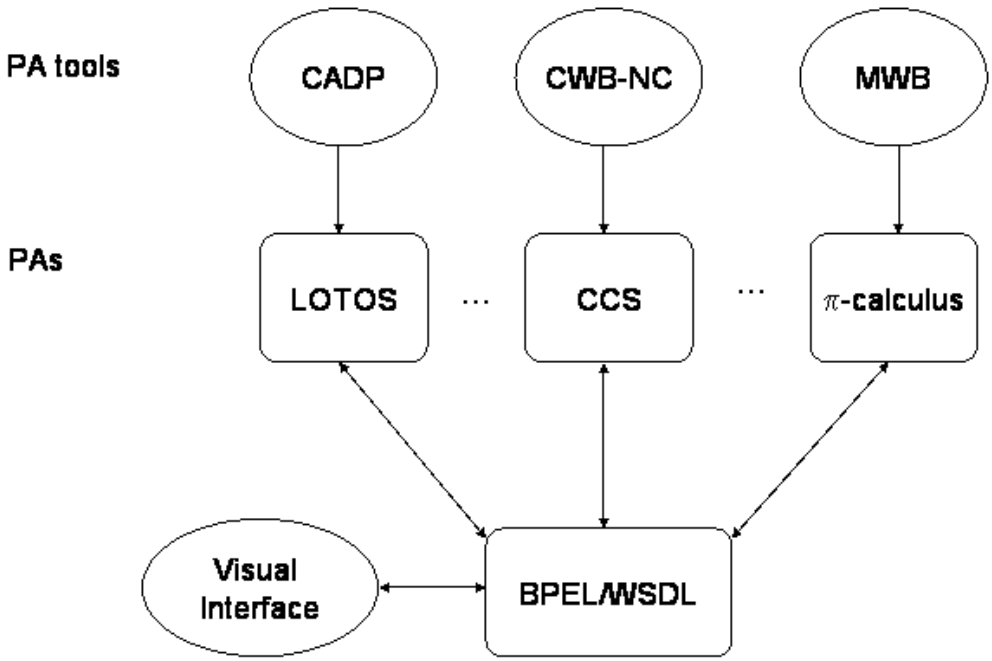

Figure 6.1: Proposal overview

constraints about this. In order to obtain a simple automatic translation from a process algebra, we have to follow this simple rule in the design. The details of other similar rules will be given during the explanation. We remark that in our framework, when we design and correct in BPEL, the LOTOS-BPEL direction is free from this problem: we start from LOTOS code, that is BPEL-like structured, because directly obtained by the translation from BPEL.

In our presentation we refer to Table 6.1 and to Table 6.2, where we show sample code of both languages; the correspondence is about both directions of the mapping. Figure 6.2 gives a very general picture. We show the mapping of basic construct, dynamic behavior, data definition and handling, and fault, compensation, event handlers. Finally we give general guidelines for translations between PAs and BPEL.

Sample BPEL Code	**Sample LOTOS Specification**

`< ... act1 ... >`
`</act1>`
`<assign ... >`
`<copy>`
`<from expression="5"/>`
`<to var="x"/>`
`<copy>`
`</assign>`
`< ... act2 ... >`
`</act2>`
	..act1..; exit(5) ≫
	accept x:Nat in ..act2..

`<receive ... variable="m">` `</receive`	g?m:Nat;
`<reply ... variable="m">` `</reply>`	g!m:Nat;

`<invoke ... invar="mS"`
`outvar="mR">`
`</invoke>`
	gS!mS:Nat; gR?mR:Nat;

Table 6.1: The BPEL-LOTOS two-way mapping: examples for basic behaviors.

6.2.1 General Outline

An external view of interacting WSs shows processes (services) running concurrently. Such a kind of global system in LOTOS is described using LOTOS main behavior (that is the outermost process): it instantiates processes composed in parallel and synchronizing on all actions representing their interactions.

At the basis of our mapping there is the correspondence between LOTOS actions and BPEL interactions. BPEL services and LOTOS processes instantiated in the main process correspond to each other. The direction from BPEL to LOTOS is straightforward: we simply automatically build a main behavior containing the instantiation of all the processes (each of them correspond to a service), in the manner described above. About the other direction, from LOTOS to BPEL, the LOTOS programmer have to respect this rule: he has to write the main behavior simply instantiating all the processes representing services, in the usual manner.

To describe behaviors, in LOTOS we have the process definition, in BPEL the service description. In LOTOS a defined process can be instantiated (with action passing, that renames the name of action in the definition, and parameter passing). From LOTOS to BPEL, we use the behaviors

Sample BPEL Code	Sample LOTOS Specification
<pick ... > <onMessage ... variable="m1"> < ... act1 ...> </onMessage> <onMessage ... variable="m2"> < ... act2 ... > </onMessage> </pick>	(g1?m1:Nat; ..act1..) [] (g2?m2:Nat; ..act2..)
<sequence ...> < ... act1 ... > < ... act2 ... > </sequence>	..act1..; ..act2..
<flow ... > < ... act1 ... > <source linkname="link1" condition="cond1"/> </act1> < ... act2 ... > <target linkname="link1"/> </act2> </flow>	..act1..; ([cond1]->link1 !1; [] [not(cond1)]->link1 !0;) || (link1 ?x:Bool; ([x=1]->..act2.. [] [x=0]->i;))
<switch> <case condition= "bpws:getVariableData(x)>=0"> <..act1..> </..act1..> </case> <otherwise> <..act2..> </..act2..> </otherwise> </switch>	[x>=0] -> ..act1..; [] [x<0] -> ..act2.. ;
<while condition= "bpws:getVariableData(x)>=0"> <..act1..> </..act1..> </while>	proc while1 [..](..) := [x<0]-> i; [] [x>=0]->..act1..; while1[..](..) endproc

Table 6.2: The BPEL-LOTOS two-way mapping: examples for structured behaviors.

specified in the process definition to generate the BPEL service description with the names of partner links, port type, operations, variables. From BPEL to LOTOS, we use the service description to generate, including the names of actions, both the process definition and the process instantiations. We have a process instantiation if the process represents a scope or a while.

We do not consider bindings issues. For the data type definitions in BPEL/WSDL we have XMLSchema, in LOTOS we can define abstract data types. In LOTOS we initialize the data structures defined with the type construct at the beginning of the main process.

To summarize, the main process first initializes data, then instantiates the process/services running concurrently.

6.2.2 Basic behaviors and interactions

At the core of BPEL process model is the notion of peer-to-peer interaction between partners described in WSDL. All BPEL basic activities perform interactions between WSs. An interaction is characterized by the partner link, the port type, and the operation involved in the two communicating partners (each partner defines these three elements for each interaction). In parallel, LOTOS has at its disposal the notion of action to represent dynamic evolutions and of rendez-vous to describe synchronizations among processes. Consequently, when process/services are instantiated, LOTOS synchronizing actions are equivalent to BPEL interactions. When the process representing a service is defined, an action is simply an emission or a reception. The name of the action stores information (parter link, port type, operation in BPEL, process and action names in LOTOS) on the receiver in the emission case, on the sender in the reception case. This name can contain a description of the interaction (e.g. request, notification, cancellation). When we instantiate, we have to compose the names of the action of both interacting processes/services; we consider two synchronizing action, we concatenate their definition name, and we give the concatenated name to both.

Let us go forward in more details. Starting the mapping from BPEL, in order to build the name of LOTOS action, we use the information in *partner link*, *port type*, *operation* attributes in the *receive*, *reply*, and *invoke*. Let a partner 1 (resp. 2) have a partner link pl_1 (resp. pl_2), a port type p_1 (resp. p_2), an operation o_1 (resp. o_2), and a variable v_1(resp. v_2) associated with the exchanged message. Let a reservation request the object of the partner 1 message, and an availability response the object of the partner 2 message. Then the process associated with the partner 1 has *in the definition* the action $pl_1\_p_1\_o_1\_resReq$ and the process for the partner 2 the action $pl_2\_p_2\_o_2\_avResp$. When the two processes are instantiated in the main behavior, the name of their synchronized action is $pl_1\_p_1\_o_1\_resReq\_pl_2\_p_2\_o_2\_avResp$, and v_1 (resp. v_2) is the parameter of the action for the partner 1 (resp. 2). Moreover if we have a *message* with N *part* tags, in LOTOS we have an action with N parameters, one for each part of the message.

Starting from LOTOS instead, we extract the port type, operation and message definitions analyzing the names of LOTOS actions in the instantiated processes. For example if we have two actions $pl_1\_p_1\_o_1\_pl_2\_p_2\_o_2$ and $pl_1\_p_1\_o'_1\_pl_2\_p_2\_o_2$, we conclude that we have the service 1 with partner link pl_1, port type p_1 and operations o_1 and o'_1, and a service 2 with partner link pl_2, port type p_2 and operation o_2. Moreover we know that there are two interactions: one between service 1 in partner link pl_1, port type p_1, operation o_1, and service 2 in partner link pl_2, port type p_2, operation o_2; the other interaction is between service 1 in partner link pl_1, port type p_1, operation o'_1, and service 2 in partner link pl_2, port type p_2, operation o_2.

The reception of a message is expressed using the *receive* activity in BPEL and using a action with a reception in all its parameters in LOTOS.

In BPEL, the emission is written with the *reply* or the *asynchronous invoke* activity whereas in LOTOS we use a action with an emission in all its parameters. The BPEL *synchronous invoke*,

performing two interactions (sending a request and receiving a response) corresponds in LOTOS to an emission followed immediately by a reception. In LOTOS we have two different actions, because we have two interactions in BPEL; the names of actions share the same partner link, the same port type, the same operation but they differs only by a letter S or R at the end (representing the emission and the reception of the invoke). Using this rule we can distinguish in the LOTOS code when a contiguous emission-reception is an invoke.

6.2.3 Structured Behaviors

Now we introduce the mapping for LOTOS dynamic constructs and BPEL structured activities.

The *pick* BPEL activity is executed when it receives one message defined in one of its *onMessage* tag or when it is fired by an *onAlarm* event; we cannot model the latter case because basic LOTOS does not have the notion of time. The equivalent construct in LOTOS is obtained using the non deterministic choice, in which the first action of each branch is a reception; it is chosen the branch whose beginning reception is performed first. In the LOTOS modelling, if we use the non deterministic choice with an emission as first action, then an automatic translation to BPEL becomes very difficult. For example the following LOTOS behavior, because the a is an emission, does not have a straightforward translation in BPEL:

```
... a!x:Nat; b?x:Nat; [] c?x:Nat; b?x:Nat;..
```

When we design in a process algebra, we have to think to BPEL code structure, in order to simplify the automatic translation.

The *sequence* activity in BPEL match with the LOTOS prefixing operator ';'.

In BPEL we have the *flow* activity, in LOTOS the full synchronization constructs '||'. Because in BPEL we cannot have interaction inside a service, therefore we do not have synchronizations in a parallel composition inside a process representing a service. The mapping about the *link* tag is more complicated, because LOTOS does not have an explicit construct of dependence relation between concurrent actions. In BPEL we specify with the *source* tag the activity that has to occur first, and with the *target* tag the dependent activity. In LOTOS we have an action for each link. These actions are put after the end of the source behavior, and before the beginning of the target one; the two behaviors synchronizes on these actions, that is they have to execute them at the same time. In this way we are sure that the source behavior is completed before the beginning of the target one. In Table 6.2, in the flow sample, activity *act2* can be executed only both after executing activity *act1* and the condition *cond1* is true; in the BPEL code, this condition is specified by the *transitionCondition* attribute. In LOTOS after executing $act1$, we execute the action $link1$, representing the link, and assign to its parameter the value 1 if the condition *cond1* is true, 0 otherwise; $act2$ can be executed only if the condition is true and only after $act1$, because it can be executed only after the action $link1$. If we cannot execute $act2$, we have to choices: just to skip $act2$ (in LOTOS we do nothing putting an i action, in BPEL we set $suppressJoinFailure = yes$), or to trow a fault (in LOTOS we put a $fault$ action, in BPEL we set $suppressJoinFailure = no$). More about the links can be found in the Section 6.2.7.

The *switch* tag defines an ordered list of *case* tag. A case corresponds to a possible activity which may be executed. The condition of a case is a Boolean expression on variables. In our process algebra we have a standard pattern combining guarded expression and non deterministic choice, very often used in the design with LOTOS.

To define an environment with own local variables and with own handler of faults and events, in BPEL there is the *scope* activity, in LOTOS the concept of local process. The process corresponding to the scope is local to the process representing the outer scope. The outermost scope in BPEL is the global one. We deal with this activity in Section 6.2.5

The *while* BPEL tag and LOTOS recursive processes correspond to each other. The condition of the *while* is the exit condition of the recursive process. The behavior of this recursive process matches exactly the body of the BPEL loop, and conversely. The recursive process is instantiated by the process corresponding to the scope that contains the while. In the LOTOS modelling, recursive processes have to respect the structure of the BPEL while, in order to simplify the translation.

6.2.4 Data Descriptions

In this subsection, we are going to discuss three levels of data representation in LOTOS and BPEL: data type definitions, XPath and LOTOS, data manipulation.

Data type definitions LOTOS allows us to define abstract data types, that is data domains and operations on them (e.g. a list with operations: add an element, extract the first one etc.); many basic types (char, natural, etc.) are already defined. In BPEL, types are described using XMLSchema; elements can be simple (lots are already defined) or complex (composed by other elements). A simple element and LOTOS basic data type corresponds each other; moreover we can use the *rename* construct in LOTOS, for example to rename the type string with 'lastname'. We have a complex element in XMLSchema and abstract data type in LOTOS, having one data type for each element composing the complex one.

In XMLSchema complex elements can be composed in different manners, depending from the indicators that establish the order, and the number of occurrences of simple elements.

Order Indicators. The indicator *all*: each element occurs exactly once, in any order. In LOTOS we can define the abstract data type *list*. The element of the list, that can be added in any order, are the element in the complex type. The indicator *choice*: an element in a set is chosen. In LOTOS we can define the abstract data type *set*. The element of the set are the element in the complex type. The indicator *sequence*: it specifies the elements and the order in which they have to appear. In LOTOS we can use a list whose elements can be added only in a fixed order, depending on the type.

Occurrence Indicators. They are use to define how often an element can occur, in details *maxOccurs* the maximum number of times and *minOccurs* the minimum. In LOTOS we have the constraints on the list with a fixed order.

Group Indicators. They define a set of elements, with indicators, that can be referenced in another element. In LOTOS we can simply use the abstract data type of the group in the abstract data type of the element that uses the group. For example, if a 'choice' is referenced in a 'all' indicator, an element of the list is a set.

Variable declaration and manipulation In LOTOS, variables are either parameters of processes or parameters of a action. In BPEL, variables can represent both data and messages. They are defined using the *variable* tag (global when defined before the activity part) and their scope may be restricted (local declarations) using a *scope* tag. In LOTOS, only process parameters need to be declared (not necessary for action variables) whereas in BPEL either global and local variables involved in interactions have to be declared. In LOTOS, in local process we can declare local variables.

A BPEL *message* corresponds to a set of action parameters in LOTOS. In particular a BPEL *part* corresponds to a parameter of a action in LOTOS.

The BPEL *assign* tag has three equivalents in LOTOS depending on their use: (i) *let* X_i:T_i=V_i *in* B means the initialization of variables X_i of types T_i with values V_i ($\forall i \in 1..n$) in the behavior B, (ii) B_1*; exit(*Y_i*)* $\gg$ *accept* X_i:T_i *in* B_2 denotes the modification of variables X_i (replaced by new values Y_i), (iii) *P(*X_i*)* is an instantiation of a process or a recursive call meaning assignments of

values X_i to the parameters of the process P. Conversely, these LOTOS constructs can be mapped into BPEL using *assign*, and more precisely the *copy* tag.

LOTOS and XPath In BPEL/WSDL we can define either message or data variables, whose type are XMLSchema data structures (element or complex element). XPath is used in BPEL to manipulate data structures: to select element in a complex one, to get value from a variable, to perform operations (e.g. sum, multiplication). LOTOS data structures are abstract data type that are endowed by operations. For example lists have operations for adding an element, extracting the first element and so on. Natural numbers has sum, multiplication and so on. We can use these operations to manipulate data structures. In BPEL we use XPath as expression language; for example we can query data from a variable, and if the variable is a complex type (e.g. a record), we can select the part of interest and retrieve the value. LOTOS instead is similar to the common programming languages like C: when we write a variable, we have its value directly, as in the assign example of Table 6.1.

6.2.5 BPEL scope and LOTOS pattern of processes

In BPEL the *scope* tag defines a behavior context (local variables, event handlers, fault handlers, compensation handler) for its primary activity. The primary activity describes the normal behavior of the scope. In LOTOS we can define a pattern of processes that behaves in the same way. We point out that a LOTOS user, in the design of a scope with handlers have to respect this pattern of processes, in order to obtain automatically BPEL code. Vice versa, from BPEL specification we can get the LOTOS one, automatically filling this pattern of processes.

In BPEL we can have nested scope. The outermost scope is the global service. In LOTOS we have the concept of local process. In LOTOS the process/scope is local to the outer process/scope. Each process/scope instantiate the following processes:

primary activity: a process *primaryActivity* for the primary activity of the *scope*. In the case of normal termination, its last action is an *end* (to end fault and event handlers); we explain it below.

event handler: a process *eventHandlers*, executed in full synchronization with *primaryActivity*, because in BPEL event handlers are concurrent with the primary activity of the *scope* to which the event handler is attached.

fault handlers: a process $faultManager$ that catches a fault storing its name, launches the process *Kill* to terminate the *primaryActivity* and $eventHandlers$, then, depending on the fault name, calls the corresponding process to perform the fault activities.

compensation handler: a process for the compensation handler. In BPEL we can have at most one compensation handler in a *scope*. The name of the compensation handler is the name given in the *scope* attribute of the activity *compensate*. This process models the activity of the corresponding compensation handler.

Each process/scope has the following structure (LOTOS pseudo-code):

```
proc scopeName [..](..) :=
 ( (primaryActivity[..](..) || eventHandlers[..](..))
    [> Kill[]()
 )
 |[fault,end]| faultManager[..](..)
endproc
```

The *eventHandlers* is concurrent with *primaryActivity*; they both can be interrupted by the process *Kill*, launched by the $faultManager$ when a fault occurs. The process for the compensation handler is called inside a process representing a fault or another compensation handler: in BPEL it can be invoked, using the *compensate* tag, only either in a fault handler or in another compensation handler.

Now, we introduce the translation about the handlers in details.

Fault Handlers When a fault occurs in a BPEL scope, all activities in the primary activity and in the event handlers of the scope begin to terminate. Let $faultName$ the parameter that stores the name of the fault . In LOTOS we define a process *faultManager* running concurrently respect the process representing the scope, synchronizing on the actions *fault* and *end*. The *fault* action has the parameter $faultName$ to communicate the name of the fault; the *end* does not have parameters because we do not need to send or to receive messages, but only to communicate an event. It follows the faultManager definition:

```
proc faultManager [fault, end] (faultName:String):=
  ( fault?faultName:String;  Kill;
    [faultName1]-> faultProc1[..](..)
    [faultName2]-> faultProc2[..](..)
    ..
  )
  [] end;
endproc
```

If the scope terminates without faults, the process representing the scope performs as last action the action *end*, allowing to $faultManager$ to terminate without doing nothing. A fault in BPEL is launched through the tag *throw* (that has with attribute the name of the fault) or as response to an invoke activity; in LOTOS through action $fault$. After this the process $Kill$ is instantiated. This process doing nothing, but terminate *primaryActivity* and $eventHandlers$ using the disabling operator '[>'. Finally, the process corresponding to the fault name is chosen: for example $faultProc1$ corresponds to the fault $faultName1$.

We consider now the problem of fault propagation and handling. In BPEL, when a fault occurs in a scope S that cannot handle it, S terminates abnormally and the fault is propagated to the next scope up. If S can handle the fault, it terminates normally after executing the fault handler activities. From BPEL to LOTOS translation we know which fault handler will catch a fault by parsing the BPEL files. Similarly, from LOTOS to BPEL translation, by parsing the LOTOS specification we know the fault handler that will catch the fault; if the fault is not caught, we have a *stop* action instead of the *fault* one.

Compensation Handlers While a business process is running, it might be necessary to undo one of the steps that have already been successfully completed. To each scope we can optionally associate its compensation handler that undoes the primary activity of the scope; once a scope completes successfully, its compensation handler become ready to run. This can happen in either of two cases: explicit or implicit compensation. We map the compensation handler into a LOTOS process local to the process representing the scope.

explicit compensation: It occurs upon the execution of a *compensate* activity, that can occur inside a fault handler or a compensation handler of the scope immediately enclosing the scope to be compensated; the *compensate* activity has an attribute *scope* whose value specifies the

name of the scope to be compensated. The *compensate* activity is modelled in LOTOS by a call to the process representing the compensation handler associated with the scope.

implicit compensation: It occurs when there is a fault handling. Let A be a scope, and B an its nested compensatable scope. Consider the following scenario: B is completed successfully, but another activity in A throws a fault. Implicit compensation ensures that whatever happened in scope B get undone by running its compensation handler. Therefore, the implicit compensation of a scope goes through all its nested scopes and runs their compensation handlers in reverse order of completion of those scopes. We can map this mechanism in LOTOS by calling in the same order the processes representing the compensation handlers; all these calls are executed in $faultManager$ of A before the beginning of the fault activities. Obviously we have to store the order in which the scopes are completed. We can use a queue data structure in LOTOS to do it; this queue has global visibility and it is updated when a scope completes or if a scope is compensated. The process $faultManager$ of A can use this structure to know the order of completion.

Event Handlers We have to consider the BPEL semantics of the event handler: it can accept messages an arbitrary number of times, until the scope ends. We adopt in LOTOS a recursive process, concurrent to the primary activity of the scope in which is contained. We cannot model the *onAlarm* tag, because in LOTOS there is no notion of time. It follows the structure of $eventHandlers$:

```
proc eventHandlers [onMessage1, onMessage2,..](..):=
  ( (
      (onMessage1?m1:T; ..act_m1..;) []
      (onMessage2?m2:T; ..act_m2..;) []
       ..
    )
    eventHandlers [onMessage1, onMessage2,..](..);
  )
  [] end;
endproc
```

The action *onMessage1* represents the reception of a message $m1$, whose type is T. After receiving the message, the corresponding activity *act_m1* is executed. Then the process recursively calls itself, and it ends when an *end* interaction happens.

6.2.6 Guidelines for translation between a PA and BPEL

Slightly modifying the mapping for LOTOS, we easily obtain a mapping for other process algebras. In fact, while syntactically different, they share many concepts: the emission (message sending), the reception (message receiving), the sequence of actions, the concurrency of actions (parallel composition) and their synchronization, the processes and local ones, non deterministic choice of actions. In Figure 6.2 we give the outline of the correspondences. We remark that for modelling in a PA, if one wants a simple automatic translation, the PAs processes have to respect the BPEL structure, as in LOTOS.

If the PA does not support the data definition and handling, the mapping is slightly different: in this case the messages are tokens, and we cannot distinguish between parts in a message. In details from BPEL to PAs we use the information in *partner link*, *port type*, *operation* attributes in order to build the name of a action. If a partner 1 (resp. 2) has a partner link pl_1 (resp. pl_2), a

BPEL concept	**Process Algebra concept**
service (process)	process
scope	local process
interaction	synchronizing action
receive	reception
reply	emission
asynchronous invoke	emission
synchronous invoke	emission immediately followed by a reception
sequence	sequence construct
flow	parallel composition
while	recursive process
pick	non deterministic choice

Figure 6.2: The BPEL-PA correspondences

port type p_1 (resp. p_2), an operation o_1 (resp. o_2), and a variable v_1(resp. v_2) associated with the exchanged message, then the name of the action in the instantiation is $pl_1\_p_1\_o_1\_v_1\_pl_2\_p_2\_o_2\_v_2$. In another words, now the message it is not a parameter of the action, but it is a part of the action name: it characterizes the interaction. It is worth noting that for the translation from PAs to BPEL, if the designer respects such a structure, partner links, port types, and operations involved in the BPEL interactions can be deduced automatically from PAs actions. Otherwise, the user have to give the names manually.

We end this section discussing why LOTOS is a better choice than other process algebras. A first advantage is due to the disabling operator: if a PA does not have a disabling operator (e.g. CCS, π-calculus), it is much more complex and inefficient (but still possible) to deal with the BPEL handlers.

Another advantage of LOTOS is the data definition and the data handling. We can verify services that deals with data and with messages having more than one part, about properties that depends on values; we can carry out a black box testing. Moreover, starting the modelling from LOTOS, we can check the data types.

6.2.7 An Example

Now we give an example of translation using the "Loan Approval Process", taken from [4]. The service interacts with the services of customer, loan assessor, loan approver. A loan amount is proposed by the customer. If the amount is lower than $10000, and if the loan assessor gives a "low-risk" assessment, the loan is approved; otherwise the approver have to make the decision. In any case the service communicates to the customer the decision.

If follows the WSDL message definition:

```
<message name="creditInformationMessage">
   <part name="firstName" type="xsd:string"/>
   <part name="name" type="xsd:string"/>
   <part name="amount" type="xsd:integer"/>
```

```
</message>

<message name="approvalMessage">
   <part name="accept" type="xsd:string"/>
</message>

<message name="riskAssessmentMessage">
   <part name="level" type="xsd:string"/>
</message>

<message name="errorMessage">
   <part name="errorCode" type="xsd:integer"/>
</message>
```

In the business process defined below, the interaction with the customer is represented by the initial `<receive>` and the matching `<reply>` activities. The use of risk assessment and loan approval services is represented by `<invoke>` elements. All these activities are contained within a `<flow>`, and their (potentially concurrent) behavior is staged according to the dependencies expressed by corresponding `<link>` elements. Note that the transition conditions attached to the `<source>` elements of the links determine which links get activated. Because the operations invoked can return a fault of type "loanProcessFault", a fault handler is provided. When a fault occurs, control is transferred to the fault handler, where a `<reply>` element is used to return a fault response of type "unableToHandleRequest" to the loan requester. We omit the details in the code not involved in the translation.

```
<process name="loanApprovalProcess"
         suppressJoinFailure="yes" ...>
 ...
<!-- variables declaration>
<variables>
 <variable name="request"
    messageType="lns:creditInformationMessage"/>
 <variable name="risk"
    messageType="lns:riskAssessmentMessage"/>
 <variable name="approval"
    messageType="lns:approvalMessage"/>
 <variable name="error"
    messageType="lns:errorMessage"/>
</variables>

<!-- fault handler definition>
<faultHandlers>
 <catch faultName="lns:loanProcessFault"
        faultVariable="error">
    <reply partnerLink="customer"
           portType="lns:loanServicePT"
           operation="request"
           variable="error"
           faultName="unableToHandleRequest"/>
 </catch>
```

```
</faultHandlers>

<!-- behavior definition>
<flow>
  <links>
     <link name="receive-to-assess"/>
     <link name="receive-to-approval"/>
     <link name="approval-to-reply"/>
     <link name="assess-to-setMessage"/>
     <link name="setMessage-to-reply"/>
     <link name="assess-to-approval"/>
  </links>
  <receive partnerLink="customer"
           portType="lns:loanServicePT"
           operation="request"
           variable="request" createInstance="yes">
     <source linkName="receive-to-assess"
       transitionCondition=
       "bpws:getVariableData('request','amount')
                                     < 10000"/>
     <source linkName="receive-to-approval"
       transitionCondition=
       "bpws:getVariableData('request','amount')
                                     >=10000"/>
  </receive>
  <invoke  partnerLink="assessor"
           portType="lns:riskAssessmentPT"
           operation="check"
           inputVariable="request"
           outputVariable="risk">
     <target linkName="receive-to-assess"/>
     <source linkName="assess-to-setMessage"
       transitionCondition=
       "bpws:getVariableData('risk','level')='low'"/>
     <source linkName="assess-to-approval"
       transitionCondition=
       "bpws:getVariableData('risk','level')!='low'"/>
  </invoke>

  <assign>
     <target linkName="assess-to-setMessage"/>
     <source linkName="setMessage-to-reply"/>
     <copy>
        <from expression="'yes'"/>
        <to variable="approval" part="accept"/>
     </copy>
  </assign>

  <invoke  partnerLink="approver"
```

```
            portType="lns:loanApprovalPT"
            operation="approve"
            inputVariable="request"
            outputVariable="approval">
      <target linkName="receive-to-approval"/>
      <target linkName="assess-to-approval"/>
      <source linkName="approval-to-reply" />
  </invoke>
  <reply  partnerLink="customer"
          portType="lns:loanServicePT"
          operation="request"
          variable="approval">
    <target linkName="setMessage-to-reply"/>
    <target linkName="approval-to-reply"/>
  </reply>
</flow>

</process>
```

Now we show the LOTOS code. Due to the lenght of the action names, we shorten them the following way:

```
request <- customer_lns:loanServicePT_request
check_S <- assessor_lns:riskAssessmentPT_check_S
check_R <- assessor_lns:riskAssessmentPT_check_R
approve_S <-approver_lns:loanApprovalPT_approve_S
approve_R <-approver_lns:loanApprovalPT_approve_R
```

To specify parts of the message in BPEL we have the *part* tag, in LOTOS the action parameters. Each parameter of the actions models a part of the exchanged message. For example the BPEL variable `request` has three parts: `firstName` (string), `name` (string), `amount` (integer). In LOTOS we have three parameters with same corresponding names and types.

The *Approval* process definition, without considering the dependencies between activities expressed by links:

```
proc Approval [request, check_S, check_R, approve_S, approve_R]
              (firstName:String, name:String, amount:Integer,
               level:String, accept:String) :=
 \\ receive
 request ?firstName:String ?name:String ?amount:Integer

 ||

 \\invoke
 check_S !firstName:String !name:String !amount:Integer;
 check_R ?level:String

 ||

 \\ assign
```

```
 exit(yes) accept accept:String in {

   \\ invoke
   approve_S !firstName:String !name:String !amount:Integer;
   approve_R ?accept:String
   ||

   \\ reply
   request !accept:String
 }
endproc
```

It follows the *Approval* process definition, now considering the dependencies between activities expressed by links:

```
proc Approval [request, check_S, check_R, approve_S, approve_R,
               link_receive-to-assess, link_receive-to-approval,
               link_assess-to-setMessage, link_assess-to-approval,
               link_setMessage-to-reply, link_approval-to-reply]
              (firstName:String, name:String, amount:Integer,
               level:String, accept:String,
               b1:Bool, b2:Bool, b3:Bool, b4:Bool) :=

 \\receive
 request ?firstName:String ?name:String ?amount:Integer;
 ( [amount < 10000] -> link_receive-to-assess;
                       \\ source receive-to-assess
   []
   [amount >= 10000] -> link_receive-to-approval!1;
                        \\ source receive-to-approval
 )

 ||

 \\ invoke
 link_receive-to-assess;   \\ target receive-to-assess
 check_S !firstName:String !name:String !amount:Integer;
 check_R ?level:String;
 ( [level = 'low'] -> link_assess-to-setMessage;
                       \\ source assess-to-setMessage
   []
   [level != 'low'] -> link_assess-to-approval!1;
                       \\ source assess-to-approval
 )

 ||

 \\ assign
 link_assess-to-setMessage;      \\ target assess-to-setMessage
 exit(yes) accept accept:String in {
```

```
   link_setMessage-to-reply!1;   \\ source setMessage-to-reply

   \\ invoke
   \\ target receive-to-approval and assess-to-approval
   ( link_receive-to-approval ?b1:Bool   []
     link_assess-to-approval ?b2:Bool );

   ( link_receive-to-approval ?b1:Bool   []
     link_assess-to-approval ?b2:Bool );

   ( [b1=1 AND b2=1]->
        ( approve_S !firstName:String !name:String !amount:Integer;
          approve_R ?accept:String );
    []
     [not(b1=1 AND b1=1)]-> i
   )
   link_approval-to-reply!1;   \\ source approval-to-reply

   ||

   \\ reply
   \\ target setMessage-to-reply and target approval-to-reply
   (  link_setMessage-to-reply ?b3:Bool   []
      link_approval-to-reply ?b4:Bool );
    []
   (  link_setMessage-to-reply ?b3:Bool   []
      link_approval-to-reply ?b4:Bool );
   (  [b3=1 AND b4=1]-> request !accept:String
    []
      [not(b3=1 AND b4=1)]-> i
   )
 }
endproc
```

The case of an activity that depends from two or more other activities is more complicated: the source activities can happen in any sequence. In LOTOS we model this situation using the non deterministic choice in which each branch is a link activity; if an activity depends from N source activities, we repeat the non deterministic choice of all link action N times. In this way the we have to receive N values; we ensure that we receive N different aknowledgement of a source activity execution by the guarded expression before performing the target activity. For example the last `request` has to be executed after `link_setMessage-to-reply` and `link_approval-to-reply` in any order; using the non deterministic choice we ensure that we receive two aknowledgement of execution of the source activity. We ensure that both activities are performed by executing the target activity `request` only if we have $b3 = 1$ (the action `link_setMessage-to-reply` is executed) and $b4 = 1$ (the action `link_approval-to-reply` is executed). If $b3 = 0$ or $b4 = 0$ we simply do nothing, according to the $suppressJoinFailure =' yes'$ attribute in the BPEL process tag. We remark that $b3$ and $b4$ are initialized with 0 by the main process; all LOTOS parameters related with the link actions have to be initialized with 0 (we always follow this rule in the mapping about the links).

The *faultManager* process definition:

```
proc faultManager [fault, end] (faultName:String) :=
 ( fault?faultName:String; Kill;
   [lns:loanProcessFault]-> faultProc1[request](error)
 )
 [] end;
endproc
```

The *faultProc1* process definition:

```
proc faultProc1 [request](error:Integer) :=
  request !errorCode:Integer
endproc
```

Finally we give the LOTOS main process, considering also the links; we concatenate the names of two actions in process definition that constitute an interaction (e.g. the action `request_c` is the interaction composed by the action 'request' (a reception) in the process definition `Approval` and the action 'c' (an emission) in the process definition `Customer`). The two actions are instantiated with the same name. For example the `request` in the process definition `Approval` becomes, in the `Approval` instantiation, `request_c`; the action `c` in the process definition `Customer` is replaced by `request_c` in the instantiation. The actions corresponding to the links are not renamed (they do not model an interaction between services); we always follow this rule in the mapping about the links.

```
proc main [request_c, check_S_ass, check_R_ass,
           approve_S_app, approve_R_app,
           link_receive-to-assess, link_receive-to-approval,
           link_assess-to-setMessage, link_assess-to-approval,
           link_setMessage-to-reply, link_approval-to-reply, ..]
          (firstName, name, amount, level, accept, ..) :=
  ( ( Approval[request_c, check_S_ass, check_R_ass,
               approve_S_app, approve_R_app,
               link_receive-to-assess, link_receive-to-approval,
               link_assess-to-setMessage, link_assess-to-approval,
               link_setMessage-to-reply, link_approval-to-reply]
              (firstName, name, amount, level, accept, 0, 0, 0, 0)
      || Customer[request_c,..](..)
      || Assessor[check_S_ass, check_R_ass,..](..)
      || Approver[approve_S_app, approve_R_app,..](..)
    )
    [> Kill[]()
  )
  |[fault,end]|
  faultManager[]()
endproc
```

6.3 Design and verification features

In this section we discuss the features that process algebras provide for design and verification, and we sketch some problems, for a future work, that deal with simulation and bisimulation. Examples

of WS developed using process algebras can be found in [105], where a sanitary agency is modelled in CCS, and in [106], where a simple e-commerce application is designed using LOTOS.

Distributed development and reuse. Allowing the modularization, the modelling from a process algebra supports the distributed development and the software reuse.

Verification features. The following verification facilities are available at an early stage of the Web services deployment:

temporal logic model checking, in order to prove properties of the service: liveness (something good happens), safety (bad events do not happens), request-response (a request is always satisfied, also for infinite behaviors), and others. We can verify for example mutual exclusion properties (e.g. if the provider can satisfies only one request among multiple concurrent requests, it satisfies the first confirmed request). If the property is not satisfied, a counterexample is returned.

bisimulation, to check whether the behaviors of two services or two versions of the same service are equivalent; if they are different, it is shown a counterexample.

simulation, to check whether the behavior of a services is included by the behavior of other interacting services; if it is not, it is shown a counterexample.

execution traces of the service (manually or random guided), to understand the behavior of the service. In the verification community (and in [91]), often the simulation name is used to denote execution traces analysis; this is no the case of this chapter.

In the case of a process algebra allowing the data handling, it is available:

data type checking, in the case of LOTOS and other process algebras allowing data handling.

black box testing: for a class of input values, some properties are satisfied.

Respect previous approach [91, 53, 90, 55], one of the main advantage of using process algebra is the availability of the simulation and bisimulation analysis; simulation supports the hierarchical refinement design method, while the bisimulation allows the redundancy analysis of a community, and it can be used to establish when a service can substitute another one in a composition [28]. Moreover we argue, with a simple consideration, that the simulation can be part of the problem of automatic composition of services. In the rest of the section we discuss briefly these issues, considering them for a future work.

Hierarchical refinement [78, 76]. It is a well-known method for design development. It proceedes top-down: starting with a highly abstract specification, we construct a sequence of behavior descriptions, each of which refers to its predecessors as a specification, and is thus less abstract than the predecessor. At each stage the current implementation is verified to satisfy its specification. The last description in the sequence contains no abstractions, and constitutes the final implementation. The behavioral equivalence between a specification and its implementation is checked by simulation or by a trace-based equivalence. The advantage of using a two-way mapping, rather than only the direction starting from BPEL, is that we can apply hierarchical refinement also in the BPEL modelling of WS.

Automatic Composition and Redundancy. The simulation can be part of the problem of automatic composition of services: intuitively, a service is composable from a bundle of other ones, if it can be simulated by them, that is if its behaviors are contained in their behaviors.

When a community of Web services is used to compose a new service (e.g. [16]), it is useful to know which services in the community are redundant: we calculate it off-line, using bisimulation. On-line, before starting the composition algorithm, we select services avoiding that two or more equivalent services are activated.

6.4 Related Works

We are going to introduce three kinds of related works aiming at: i) specifying WSs at an abstract level using formal description techniques and reasoning on them, ii) using jointly abstract descriptions and executable languages (mainly BPEL), iii) developing WSs from abstract specifications.

At this abstract level, lots of proposals originally tended to describe WSs using semi-formal notations, especially workflows [81]. More recently some more formal proposals grounded for most of them on transition system models (LTSs, Mealy automata, Petri nets) have been suggested [68, 91, 62, 16, 80]. With regards to the reasoning issue, works have been dedicated to verifying WS description to ensure some properties of systems [55, 41, 91, 53, 90]. Summarizing these works, they use model checking to verify some properties of cooperating WSs described using XML-based languages (DAML-S, WSFL, BPEL, WSCI). Accordingly, they abstract their representation and ensure some properties using ad-hoc or well-known tools (*e.g.* SPIN, LTSA). We have a deeper look in the following of this section at proposals focusing on BPEL.

In comparison to these existing works, the strength of our alternative approach (using PA) is to work out all these issues (description, composition, reasoning) at an abstract level, based on the use of expressive (especially compared to the former proposals) description techniques and adequate tools. The compositionality property of process algebra is also very convenient in one area where composition is one of the main concern.

The second bunch of related work [55, 97, 53, 91, 115] deals with mappings between abstract and concrete descriptions of WSs. Let us emphasize that in first attempts [105, 106], we have already proposed some guidelines to map process algebra and BPEL. Nevertheless, these guidelines (for CCS and LOTOS) were not defined in details and they deal with a subset of BPEL; in this work we include in the mapping also fault, compensation, and event handlers. Two relevant related works are [53, 55]. In the first one, the authors proposed a formal approach to model and verify the composition of WSs workflows using the FSP (Finite State Processes) notation and the LTSA tool. Their paper introduces a translation of the main BPEL structured activities (sequence, switch, while, pick and flow) into FSP processes. In the second one, it is presented an approach to analyze BPEL composite web services communicating through asynchronous messages. They use guarded automata as an intermediate language from which different target languages (and tools) can potentially be employed. They especially illustrate with the use of Promela/SPIN as the formal language and the corresponding model checker.

Compared to them, our attempt is more general: (i) we show a two-way mapping, useful to develop WSs and also to reason on deployed ones (the latter direction was the single goal of mentioned related works). All other previous works give only a mapping from BPEL to a formal language. (ii) we consider in the mapping also compensation and event handlers, and we deal with fault handlers explicitly. (iii) we can verify not only temporal logic properties, but also behaviors equivalences between services using bisimulation. Using this facilities we can apply the hierarchical refinement design method to WSs, also in the BPEL modelling.

Finally, the recent proposal of Lau and Mylopoulos [79] argue the use of TROPOS as starting point of WS design, but they do not deal with verification, but requirements issues. A more general methodology, integrating the requirements analysis and the generation of BPEL code, was proposed in [97].

6.5 Concluding Remarks and Future Work

We present a framework, for the design and the verification of WSs using process algebras. We illustrate a two-way mapping between a very expressive process algebra, LOTOS, and BPEL. We give also general guidelines for translations between a process algebra and BPEL.

Process algebras allow not only temporal logic model checking, but also a simulation and bisimulation analysis; they allow a design method, hierarchical refinement [78, 76], that we can apply to WSs. In fact the two-way mapping allows us to design and verify both in process algebra and in BPEL. In Section 6.3, we sketch how simulation and bisimulation are involved in the automatic composition of services and in the redundancy check of services. In our opinion, these connections deserve to be studied in a future work, together with the generalization of the mapping to other languages and its implementation. In our current mapping, we do not consider dynamic process instantiation and correlation set. Moreover we do not tackle the problem of the dynamic choice of the partner to talk to (our interactions are established before the conversation between partners starts); for this reason we do not consider BPEL endpoint references. It is interesting to extend the mapping in these directions. Finally, we plan to experiment the use of process algebras in the methodology proposed in [97], where only temporal model checking is performed; in particular we are interested in adding the simulation and bisimulation analysis.

Chapter 7

Conclusions

The *first aim of this thesis* is to identify conditions that make tractable some verification problems. We first study the complexity of such problems under the hypothesis of some structural restrictions (e.g. treewidth). Many problems that are intractable (e.g. NP-hard, PSPACE-hard) for general graphs, are polynomial or linear-time solvable when the graph has bounded treewidth (see [27, 26, 25] for an overview). For example, constraint-satisfaction problems, which are NP-complete, are PTIME-solvable when the variable-relatedness graph has bounded treewidth [40, 54].

Our answer is negative; we identify only a problem whose complexity decreases, under the hypothesis of bounded treewidth. In Section 3.2, we prove the positive result: the *Image Computation* complexity on a transition system with $|V|$ states and treewidth at most k is $O(|V|^2 \cdot k)$; in the case of unbounded treewidth is $O(|V|^3)$.

Then, we show a sequence of negative theorems. About *Bounded Model Checking*, one of the main approaches to Model Checking, the unrolling does not preserve treewidth (Section 3.3). In Section 3.4, we show that, even if we assume bounded pathwidth and bounded degree in the transition sysyem, *Temporal Logic Model Checking* remains PSPACE-complete, *Simulation* remains EXPTIME-complete, and *Containment* remains EXPSPACE-complete.

In Chapter 4, about Model Checking, we show that preprocessing either the transition system or the property in a polynomial size data structure - using any amount of time, does not decrease the worst case complexity. The preprocessing of the system is useful when we want to verify the same system against many properties; while the preprocessing of the property can be used when we agree on some requirements and we want to verify them against many systems. Many model checking tools support these functions (e.g. databases of property in the case of preprocessing the property, the compilation of the system in BDDs in the case of the preprocessing of the model).

Then, we *formally* prove that the superpolynomial growth in Symbolic Model Checking is unavoidable, for a large class of data structure (e.g. BDDs, ADDs). Symbolic Model Checking is one of the most effective approaches to this problem.

The *second aim of the thesis* is to show the complexity of many problems related with reachability. Reachability is an important problem both for formal verification community (it is a particular case of model checking) and for reasoning about actions area (it can be the goal of the planning problem). We discuss these results in Chapter 5.

The *third aim of the thesis* is to apply formal verification to Web services, an emerging technology in computer science. In Chapter 6, we present a two-way mapping between process algebra (a formal verification formalism) and BPEL (the main WS coordination language).

We remark the applications of this mapping: model checking analysis, the use of simula-

tion/bisimulation for verification, for the hierarchical refinement design method, for the service redundancy analysis in a community, and for replacing a service with another one in a composition.

Bibliography

[1] *XMLSchema*, www.w3c.org/XML/Schema.

[2] *XPath*, www.w3c.org/TR/xpath.

[3] P.A. Abdullah, P. Bjesse, and N. Een, *Symbolic reachability analisys based on SAT-solvers*, Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analisys of Systems (TACAS'00), 2000.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *Specification: Business Process Execution Language for Web Services Version 1.1*, 2003, http://www-106.ibm.com/developerworks/library/ws-bpel/.

[5] A. Atserias, P.G. Kolaitis, and Moshe Y. Vardi, *Constraint propagation as a proof system*, CP 2004, 2004, pp. 77–91.

[6] F. Bacchus and Q. Yang, *The downward refinement property*, Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91), 1991, pp. 286–293.

[7] C. Bäckström, *Equivalence and tractability results for SAS+ planning*, Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning (KR'92), 1992, pp. 126–137.

[8] C. Bäckström and P. Jonsson, *Planning with abstraction hierarchies can be exponentially less efficient*, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95), 1995, pp. 1599–1605.

[9] C. Bäckström and B. Nebel, *On the computational complexity of planning and story understanding*, Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), 1992, pp. 349–353.

[10] ———, *Complexity results for SAS+ planning.*, Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93), 1993, pp. 1430–1435.

[11] ———, *Complexity results for SAS+ planning*, Computational Intelligence **11** (1995), 625–656.

[12] J. C. M. Baeten and W. P. Weijand, *Process algebra*, Cambridge Tracts in Theoretical Computer Science, vol. 18, Cambridge University Press, Cambridge, England, 1990.

[13] R.I. Bahar, E.A. Frohm, C.M. Gaona, C.M. Hachtel, G.D. Macii, and F. Somenzi, *Algebraic decision diagrams and their applications*, Proceedings of the International Conference CAD, 1993, pp. 188–191.

[14] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli, *Methodology and system for practical formal verification of reactive hardware*, Proc. 6th Conference on Computer Aided Verification (CAV'94) (Stanford), Lecture Notes in Computer Science, vol. 818, Springer, June 1994, pp. 182–193.

[15] I. Beer, S. Ben David, and A. Landver, *On the fly model checking for rctl formulas*, Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98), LNCS, vol. 1427, Springer, 1998, pp. 184–194.

[16] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Automatic Composition of E-services That Export Their Behavior*, Proc. of ICSOC'03 (Italy) (M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, eds.), LNCS, vol. 2910, Springer-Verlag, 2003, pp. 43–58.

[17] J. A. Bergstra, A. Ponse, and S. A. Smolka (eds.), *Handbook of Process Algebra*, Elsevier, 2001.

[18] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, *Weak, strong, and strong cyclic planning via symbolic model checking*, Artificial Intelligence.

[19] _____, *MBP: a model based planner*, IJCAI'01 Workshop on Planning under Uncertainty, AAAI Press, 2001.

[20] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso, *A framework for planning with extended goals under partial observability*, ICAPS'03, AAAI Press, 2003.

[21] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, *Planning in nondeterministic domains under partial observability via symbolic model checking*, IJCAI'01, AAAI Press, 2001.

[22] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu, *Symbolic model checking using SAT procedures instead of BDDs*, Proc. 36th Design Automation Conference, IEEE Computer Society, 1999, pp. 317–320.

[23] A. Biere, A. Cimatti, E.M. Clarke, and Yunshan Zhu, *Symbolic model checking without BDDs*, Proceedings of the 5th Int. Conf. Tools and Algorithms for the Construction and Analisys of Systems (TACAS'1999), LNCS, vol. 2031, Springer, 1999, pp. 193–207.

[24] P. Bjesse, J.H. Kukula, R.F. Damiano, T. Stanion, and Y. Zhu, *Guiding sat diagnosis with tree decompositions*, SAT 2003, LNCS, vol. 2919, Springer, 2004, pp. 315–329.

[25] H.L. Bodlaender, *A tourist guide through treewidth*, Acta Cybernetica **11** (1993), 1–23.

[26] _____, *Treewidth: Algorithmic techniques and results*, Mathematical Foundations of Computer Science 1997, 22nd International Symposium, MFCS'97, Bratislava, Slovakia, August 25-29, 1997, Proceedings (Igor Prívara and Peter Ruzicka, eds.), Lecture Notes in Computer Science, vol. 1295, Springer, 1997.

[27] _____, *A partial k-arboretum of graphs with bounded treewidth*, Tech. report, Universiteit Utrecht, 1998.

[28] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella, *When are 2 web services compatible?*, Proc. of VLDB-TES'04, LNCS, Spinger, 2004, To appear.

[29] R.K. Brayton, G.D. Hachtel, A. Sangiovanni Vincetelli, F. Somenzi, A. Aziz, S.T. Cheng, S. Edwards, S. Khatri, T. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa, *VIS: a system for verification and syntesis*, Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96), LNCS, vol. 1102, Springer, 1996, pp. 428–432.

[30] R.E. Bryant, *Graph-based algorithms for boolean-function manipulation*, IEEE Trans. on Computers **C-35** (1986), no. 8.

[31] ———, *On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication*, IEEE Transactions on Computers **40** (1991), 205–213.

[32] T. Bylander, *Complexity results for planning*, Proceedings of the 12th International Joint Conference on Artificial Intelligence (San Mateo, CA), LNCS, Morgan Kaufmann, 1991, pp. 274–279.

[33] M. Cadoli, F.M. Donini, P. Liberatore, and M. Schaerf, *Space efficency of propositional knowledge representation formalisms*, Journal of Artificial Intelligence Research **13** (1999), 25–64.

[34] ———, *The size of a revised knowledge base*, Artificial Intelligence **115** (2000), 1–31.

[35] ———, *Preprocessing of intractable problems*, Information and Computation **176** (2002), 89–120.

[36] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri, *NuSMV 2.1 user's manual*, IRST, http://nusmv.irst.itc.it, 2002.

[37] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV 2: An opensource tool for symbolic model checking*, Proceedings of the 14th International Conference on Computer-Aided Verification (CAV'02), 2002.

[38] E. Clarke, M. Fujita, P. McGeer, K.L. McMillan, J. Yang, and X. Zhao, *Multi terminal binary decision diagrams: An efficient data structure for matrix representation*, Proceedings of the International Workshop on Logic and Synthesis, 1993, pp. 1–15.

[39] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model checking*, MIT Press, 2000.

[40] R. Dechter and J. Pearl, *Network-based heuristics for constraint-satisfaction problems*, Artificial Intelligence **34** (1987), 1–38.

[41] A. Deutsch, L. Sui, and V. Vianu, *Specification and Verification of Data-driven Web Services*, Proc. of PODS'04 (Paris) (ACM, ed.), ACM Press, 2004, pp. 1–14.

[42] R. Diestel, *Graph theory*, Graduate Texts in Mathematics, no. 173, Springer-Verlag, 2000.

[43] R. Drechsler and D. Sieling, *Binary decision diagrams in theory and practice*, International Journal on Software Tools for Technology Transfer (STTT) **3** (2001), no. 2, 112–136.

[44] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag, New-York, 1985.

[45] E.A. Emerson and J.Y. Halpern, *Sometimes and not never revisited: On the branching versus linear time*, Journal of the ACM **33(1)** (1986), 151–178.

[46] A. Ferrara, *Web services: a process algebra approach*, Proc. of 2nd International Conference on Service Oriented Computing (ICSOC'04), ACM Press, pp. 242–251.

[47] A. Ferrara, P. Liberatore, and M. Schaerf, *The complexity of checking action redundancy*, AI*IA 2005: Advances in Artificial Intelligence, 9th Congress of the Italian Association for Artificial Intelligence, Lecture Notes in Artificial Intelligence, no. 3673.

[48] _____, *Model checking, preprocessing, and BDD size*, Prel. Proc. Advances in Modal Logic (AiML 2004), Manchester University Technical Report.

[49] A. Ferrara, G. Pan, and M.Y. Vardi, *Treewidth in verification: Global vs. local*, LPAR 2005, LNCS, vol. 3835, Springer, 2005, pp. 489–503.

[50] R. Fikes and N. Nilson, *STRIPS: a new approach to the application of theorem proving to problem solving*, Artificial Intelligence **2** (1971), 189–209.

[51] E. Fink and Q. Yang, *Formalizing plan justifications*, Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence, 1992, pp. 9–14.

[52] _____, *A spectrum of plan justifications*, Proceedings of the AAAI 1993 Spring Symposium, 1993, pp. 29–33.

[53] H. Foster, S. Uchitel, J. Magee, and J. Kramer, *Model-based Verification of Web Service Compositions*, Proc. of ASE'03 (Canada), IEEE Computer Society Press, 2003, pp. 152–163.

[54] E.C Freuder, *Complexity of k-tree structured constraint satisfaction problems*, Proc. AAAI-90, 1990, pp. 4–9.

[55] X. Fu, T. Bultan, and J. Su, *Analysis of Interacting BPEL Web Services*, Proc. of WWW'04 (USA), ACM Press, 2004.

[56] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W.H. Freeman and Company, San Francisco, Ca, 1979.

[57] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, Chapman & Hall, 1995, pp. 3–18.

[58] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: Theory and practice*, Morgan Kaufmann, 2004.

[59] R. Goering, *Model checking expands verification's scope*, Electronic Engineering Today (1997).

[60] R. Goldman, C. Geib, and C. Miller, *A new model of plan recognition*, Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI'99), 1999, pp. 245–254.

[61] G. Gottlob and R. Pichler, *Hypergraphs in model checking: Acyclicity and hypertree-width versus clique-width*, SIAM Journal on Computing **33** (2004), no. 2, 351–378.

[62] R. Hamadi and B. Benatallah, *A Petri Net-based Model for Web Service Composition*, Proc. of ADC'03 (Australia) (K.-D. Schewe and X. Zhou, eds.), CRPIT, vol. 17, Australian Computer Society, 2003.

[63] S. Hanks and D. Weld, *A domain-independent algorithm for plan adaptation*, Journal of Artificial Intelligence Research **2** (1995), 319–360.

[64] D. Harel, O. Kupferman, and M.Y. Vardi, *On the complexity of verifying concurrent transition systems*, Information and Computation **173** (2002), no. 2, 143–161.

[65] R.H. Herdin, Z. Har'el, and R.P. Kurshan, *Cospan*, Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96), LNCS, vol. 1102, Springer, 1996, pp. 423–427.

[66] C. A. R. Hoare, *Communicating sequential processes*, Prentice-Hall, 1984.

[67] G.J. Holzmann, *The model checker SPIN*, IEEE Transactions on Software Engineering **23(5)** (1997), 279–295.

[68] R. Hull, M. Benedikt, V. Christophides, and J. Su, *E-Services: a Look Behind the Curtain*, Proc. of PODS'03 (USA) (ACM, ed.), ACM Press, 2003, pp. 1–14.

[69] IBM Haifa, http://vlsi.colorado.edu/∼vis/usrDoc.html, *RuleBase user's manual*, 2003.

[70] ISO, *LOTOS: a Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, Tech. Report 8807, International Standards Organisation, 1989.

[71] M. Jurdziński, *Deciding the winner in partity games is in UP ∩ co-UP*, **68** (1998), 119–124.

[72] S. Kambhampati, *On the utility of systematicity: Understanding tradeoffs between redundancy and commitment in partial-ordering planning*, Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93), 1993, pp. 1380–1387.

[73] S. Kambhampati and J. Hendler, *A validation-structure-based theory of plan modification and reuse*, Artificial Intelligence **55** (1992), 193–258.

[74] C. Knoblock, J. Tenenberg, and Q. Yang, *Characterizing abstraction hierarchies for planning*, Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91), 1991, pp. 692–697.

[75] O. Kupferman, M.Y. Vardi, and P. Wolper, *An automata theoretic approach to branching-time model checking*, Journal of ACM **47(2)** (2000), 312–360.

[76] R.P. Kurshan, *Computer aided verification of coordinating processes*, Princeton Univ. Press, 1994.

[77] U. Dal Lago, M. Pistore, and P. Traverso, *Planning with a language for extended goals*, AAAI'02, AAAI Press, 2002.

[78] S.S. Lam and A.U. Shankar, *Protocol verification via projection*, IEEE Trans. on Software Engineering **10** (1984), 325–342.

[79] D. Lau and J. Mylopoulos, *Designing Web Services with Tropos*, Proc. of ICWS'04 (San Diego, USA), IEEE Computer Society Press, 2004.

[80] A. Lazovik, M. Aiello, and M. P. Papazoglou, *Planning and Monitoring the Execution of Web Service Requests*, Proc. of ICSOC'03 (Italy) (M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, eds.), LNCS, vol. 2910, Springer-Verlag, 2003, pp. 335–350.

[81] F. Leymann, *Managing Business Processes via Workflow Technology*, Tutorial at VLDB'01, Italy, 2001.

[82] P. Liberatore, *On non-conservative plan modification*, Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98), 1998, pp. 518–519.

[83] P. Liberatore, *Monotonic reductions, representative equivalence, and compilation of intractable problems*, Journal of ACM **48** (2001), no. 6, 1091–1125.

[84] Paolo Liberatore, *On the complexity of case-based planning*, Tech. Report cs.AI/0407034, 2004.

[85] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*, Springer Verlag, 1995.

[86] K.L. McMillan, *Symbolic model checking*, Kluwer Academic Publishers, 1993.

[87] R. Milner, *An algebraic definition of simulation between programs*, Proc. 2nd International Joint Conference on Artificial Intelligence, British Computer Society, 1971, pp. 481–489.

[88] _____, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science **92** (1980).

[89] _____, *Communication and Concurrency*, International Series in Computer Science, Prentice Hall, 1989.

[90] S. Nakajima, *Model-checking Verification for Reliable Web Service*, Proc. of OOWS'02, satellite event of OOPSLA'02 (USA), 2002.

[91] S. Narayanan and S. McIlraith, *Analysis and Simulation of Web Services*, Computer Networks **42** (2003), no. 5, 675–693.

[92] B. Nebel and C. Bäckström, *On the computational complexity of temporal projection, planning, and plan validation.*, Artificial Intelligence **66** (1994), no. 1, 125–160.

[93] B. Nebel, Y. Dimopoulos, and J. Koehler, *Ignoring irrelevant facts and operators in plan generation.*, Proceedings of the Fourth European Conference on Planning (ECP'97), 1997, pp. 338–350.

[94] B. Nebel and J. Koehler, *Plan reuse versus plan generation: A theoretical and empirical analysis*, Artificial Intelligence **76** (1995), 427–454.

[95] J. Obdržálek, *Fast mu-calculus model checking when tree-width is bounded*, CAV'03, LNCS, vol. 2725, Springer, 2003, pp. 80–92.

[96] J. Parrow, *An Introduction to the π-Calculus*, Handbook of Process Algebra, ch. 8, pp. 479–543, Elsevier, 2001.

[97] M. Pistore, M. Roveri, and P. Busetta, *Requirements-Driven Verification of Web Services*, Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04) (Italy), 2004.

[98] M. Pistore and P. Traverso, *Planning as model checking for extended goals in non-deterministic domains*, IJCAI'01, AAAI Press, 2001.

[99] A. Pnueli, *The temporal logic of programs*, Proceeding of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77), 1977, pp. 46–57.

[100] M.R. Prasad, P. Chong, and K. Keutzer, *Why is ATPG easy?*, Proc. of 36th ACM/IEEE conference on Design automation, ACM Press, 1999, pp. 22–28.

[101] A.N. Prior, *Past, present, and future*, Clarendon Press, Oxford, 1967.

[102] N. Robertson and P.D. Seymour, *Graph minors. i. excluding a forest*, Journal of Combinatorial Theory Series B **35** (1983), 39–61.

[103] ———, *Graph minors. ii. algorithmic aspects of treewidth*, Journal of Algorithms **7** (1986), 309–322.

[104] D. J. Rose, *On simple characterization of k-trees*, Discrete Mathematics **7** (1974), 317–322.

[105] G. Salaün, L. Bordeaux, and M. Schaerf, *Describing and Reasoning on Web Services using Process Algebra*, Proc. of ICWS'04 (San Diego, USA), IEEE Computer Society Press, 2004.

[106] G. Salaün, A. Ferrara, and A. Chirichiello, *Negotiation among Web Services using LOTOS/CADP*, Proc. of ECOWS'04, LNCS, vol. 3250, Spinger, 2004.

[107] T. Schiex, *A note on CSP graph parameters*, Tech. Report 1999/03, INRIA, 1999.

[108] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe, *Timed CSP: Theory and Practice*, Proc. of REX Workshop on Real-Time: Theory in Practice (Germany) (J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, eds.), Lecture Notes in Computer Science, vol. 600, Springer, 1992, pp. 640–675.

[109] Ph. Schnoebelen, *The complexity of temporal logic model checking*, Proceedings of the 4th Internationa Workshop in Advances in Modal Logic (AiML'02) (San Mateo, CA), vol. 4, World Scientific Publishing, 2002, pp. 1–44.

[110] A.P. Sistla and E.M. Clarke, *The complexity of propositional linear temporal logics*, Journal of ACM **32(3)** (1985), 733–749.

[111] L. J. Stockmeyer, *The polynomial-time hierarchy*, Theoretical Computer Science **3** (1976), 1–22.

[112] D.M. Thilikos, M.J. Serna, and H.L. Bodlaender, *A polynomial time algorithm for the cutwidth of bounded degree graphs with small treewidth*, ESA'01, LNCS, vol. 2161, Springer, 2001, pp. 380–390.

[113] P. Traverso and M.Pistore, *Automated composition of semantic web services into executable processes*, Proc. Third Int. Semantic Web Conference (ISWC2004), 2004, pp. 29–33.

[114] T. Villa, G. Swarny, and T. Shiple, *VIS user's manual*, VIS Group, http://vlsi.colorado.edu/~vis/usrDoc.html, 2003.

[115] M. Viroli, *Towards a Formal Foundation to Orchestration Languages*, Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04) (Italy), 2004.

[116] D. Wang, E.M. Clarke, Y. Zhu, and J. Kukula, *Using cutwidth to improve symbolic simulation and boolean satisfiability*, IEEE International High Level Design Validation and Test Workshop (HLDVT 2001), 2001, p. 6.

[117] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta, *Combining decision diagrams and SAT procedures for efficient symbolic model checking*, Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2000), LNCS, vol. 1855, Springer, 2000, pp. 124–138.

[118] Q. Yang and J. Tenenberg, *ABTWEAK: Abstracting a nonlinear, least commitment planner*, Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI'90), 1990, pp. 204–209.