# Implementation and Analysis of Behaviour Composition Problem using Simulation

A minor thesis submitted in partial fulfilment of the requirements for the degree of Masters of Computer Science

Nitin Kumar Yadav
School of Computer Science and Information Technology
Science, Engineering, and Technology Portfolio,
Royal Melbourne Institute of Technology
Melbourne, Victoria, Australia

November 30, 2009

# Declaration

This thesis contains work that has not been submitted previously, in whole or in part, for any other academic award and is solely my original research, except where acknowledged.

This work has been carried out between July and October 2009, under the supervision of Dr. Sebastian Sardina.

Nitin Kumar Yadav
School of Computer Science and Information Technology
Royal Melbourne Institute of Technology
November 4, 2009

# Acknowledgements

I am thankful to my supervisor, Dr. Sebastian Sardina, whose encouragement, guidance and support enabled me to develop an insight into the subject.

I thank Prof. Giuseppe De Giacomo and his students for allowing us to use their web-service composition systems Symfony and Opus.

I would like to thank Thomas Stroder and Maurice Pagnucco for not only providing the implementation of their approach, but also for their timely support and clarifications wherever required.

# Contents

# List of Figures

# List of Tables

# Abstract

The behavior composition problem involves realising a complex virtual behaviour by combining and coordinating a given set of existing behaviours. A behaviour in this context represents the logic of a machine, a service, a standalone component, etc. We do not have full control of the inner mechanisms of the existing behaviors, they are partially controllable. The behaviour composition problem intends to take these existing behaviours and combine them in way so as to *realize* another new complex behaviour, which may not be achieved with either of the existing behaviours. The problem is appealing to several areas of Artificial Intelligence and Computer Science, such as web-service composition, robot ecologies, ambient intelligence, agent coordination, etc. A simulation-based regression approach has been recently suggested for the behaviour composition problem. The approach has put forward a sound formal basis without a complete implementation. This minor thesis aims at providing a complete and efficient implementation of the simulation-based regression approach together with some optimisations to the originally suggested algorithm, and finally, developing a benchmarking framework for empirical evaluation.

# Chapter 1

# Introduction

The behavior composition problem involves realising a target behavior in a fully observable environment from a collection of available behaviors. A behavior in this setting models a component, service, device, software, etc. For example, day to day devices which we encounter in our lives, e.g., camera, microwave, car, mobile phone, washing machine, automatic cleaner, are all examples of a behavior. These behaviors operate in a shared environment which supports their operations, e.g., in order to wash clothes, the washing machine may consume water from the environment. A target behavior can be seen as a desired functionality that is required to be achieved.

For example, let us say there are two different kinds of mobile phones. One of them has a camera that allows it to take photographs and send messages. The other mobile has the capability to communicate through Internet, but lacks a camera. Suppose, the target behavior to be achieved requires, both taking of photographs and communicating through Internet. Then, just using one of the available mobiles would not give us the target functionality. In other words, we need to use both the available mobiles in order to *realise* the target behavior.

In our setting, we allow the available behaviors to be non-deterministic. Non-determinism means that the behaviors can evolve in more than one way after doing an action. For example, after taking a photograph, if the memory becomes full, the mobile phone may not take more photographs until memory is freed. However, after taking a photograph, if there is still space, then the mobile can keep taking more photographs. Therefore, after a taking a photograph the mobile phone can evolve in two ways: it can have two states full and not-full. Though, the phone may evolve in different ways, after its actual evolution, we know how it evolved. That is, if the memory becomes full and it cannot take any more photographs, one can observe this. We call this *full observability*, i.e., we always know the current status of a behavior.

We abstract the available and target behaviors, as well as the environment, as finite transition systems. A transition system has states and transitions. Actions cause a transition system to move through different states. For example, turning the mobile phone on is an action, that causes the phone to transition from its 'off' state to its 'on' state.

The solution to the behavior composition problem consists of coming up with a so-called *controller* that will effectively coordinate the existing behaviors such that, collectively, they act like the target behavior. The existence of such a controller can be checked by various techniques like Propositional Dynamic Logic [De Giacomo and Sardina, 2007], synthesis using Linear Temporal Logic [Sardina and De Giacomo, 2008] or alternating-time temporal logic [Felli, 2008], simulation-based regression [Sardina et al., 2008], and search-based progression [Stroeder and Pagnucco, 2009].

The simulation-based regression technique suggested by Sardina, Patrizi & De Giacomo [Sardina et al., 2008] uses the formal notation of simulation [Henzinger et al., 1995; Tan and Cleaveland, 2001] to automatically synthesise the controller, if one exists. Broadly, this approach first constructs the so-called enacted system and the enacted target. The enacted system represents all that can be achieved by the behaviors collectively when placed in the environment. Similarly, the enacted target represents the target's capabilities in the environment. The question to be asked then is, can the enacted system behave like the enacted target? If the enacted system behaves like the enacted target we say that the enacted system simulates the enacted target.

The search-based progression approach by Stroeder & Pagnucco [Stroeder and Pagnucco, 2009] follows a strategy akin to human reasoning. It checks if the available behaviors can continuously perform actions as per the target behavior. It does this by constructing search paths which lead to conditions such that the available behaviors can act like the target, and deletes the search paths which lead to opposite conditions. The search-based progression approach is backed by a prototype implementation.

In this minor thesis we focus on the simulation-based regression technique. Our contributions are as follows:

1. *We provide a complete and efficient implementation of the simulation-based regression approach.*[1]

2. *We develop a set of optimisations that can be applied to the original technique.*

[1] We are aware of two partial implementations, Symfony and Opus, for the web-services composition domain.

3. *We design a benchmarking framework for the behavior composition domain based on existing problems.*

The rest of the thesis is organized as per the following chapters:

- In chapter 2 we formally describe the behavior composition problem.
- In chapter 3 we discuss various approaches to solve the composition problem. In particular, we detail the simulation-based regression approach and briefly describe the search-based progression technique.
- In chapter 4 we describe the core components of our implementation for the simulation-based regression approach, followed by, a set of three optimisations for this approach.
- In chapter 5 we develop a benchmarking framework to allow for the empirical evaluation of the various approaches for behavior composition. We suggest a series of modular variations which can be applied to existing problems to create more complex problems.
- In chapter 6 we present empirical results for the simulation-based regression technique and its optimisations. In doing so, we briefly compare our implementation with the other known partial implementations of the regression technique.
- In chapter 7 we summarise the thesis with pointers to potential future work in this area.

# Chapter 2

# Behavior Composition Problem

The behavior composition problem has been recently addressed in AI literature [Sardina and De Giacomo, 2008; Sardina et al., 2007; 2008; Stroeder and Pagnucco, 2009]. The problem involves realising a virtual target from a given collection of available behaviors. The target is virtual in the sense that it does not exist in reality. The available behaviors need to act in a way so as to be able to conform to the target behavior's specifications. The given behaviors are controlled using a *controller*. The controller can activate a particular behavior to perform a specified action. We say a solution exists for a problem, if there exists a controller which can always guarantee the action specified by the target can always be performed by at least one of the available behaviors.

In this chapter, we formally describe the components of the behavior composition problem, followed by what makes a solution to the problem. We describe these components using the formal notations from Sardina, Patrizi & De Giacomo [Sardina et al., 2008].

## 2.1 Components: Behaviour, Environment, Target

The basic components of the behavior composition problem include the available *behaviors* acting in a shared *environment* trying to realize a virtual *target*. Formally, these components are abstracted as finite transition systems. A finite transition system consists of a number of states and transitions. The system *transitions* from one state to another when an *action* is performed. Transition systems can be categorized into a) deterministic; and b) non-deterministic. A deterministic transition system is one in which, as a result of each action, the system can evolve to just one successor state. In comparison, in a non-deterministic

system, as a result of an action, the system may evolve to more than one state. We call such an action non-deterministic. Note that in a deterministic system we always know beforehand the next state of the system given an action and its previous state. On the other hand, in a non-deterministic system we cannot predict the next state of the system for a non-deterministic action. In other words, deterministic transition systems are fully controllable whereas non-deterministic transition systems are partially controllable. We allow behaviours and environment to be non-deterministic in this scenario to allow modeling of hidden functionality, however the target to be realized is always deterministic.

### 2.1.1 Environment

An environment is a non-deterministic finite transition system $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$, where:

- $\mathcal{A}$ is a finite set of actions;
- $E$ is a finite set of environment states;
- $e_0 \in E$ is the initial state;
- $\rho \subseteq E \times \mathcal{A} \times E$ is the transition relation: $\langle e, a, e' \rangle \in \rho$, or $e \xrightarrow{a} e'$ in $\mathcal{E}$, implies that the action $a$ will cause the environment to move from state $e$ to its successor state $e'$.

If for the same action $a$ there exists more than one tuple, e.g., $e \xrightarrow{a} e'$ and $e \xrightarrow{a} e''$ then the transition system is non-deterministic. Non-determinism in the environment allows us to model incomplete information and to allow some kind of rules or decisions to be embedded in the definition.

Consider the example of the blocks-world painting domain [Sardina et al., 2008] in which there are three robot arms to *prepare*, *clean*, *paint* and *dispose* of blocks. The blocks can be cleaned or painted only after they have been prepared and after painting they need to be disposed of to allow further processing of unpainted blocks. Cleaning and painting actions consume resources like water and paint respectively. These resources might require refilling by executing the recharge action at any time. Cleaning by water is done by consuming water from the environment, whereas for painting the robot arms have their own tanks. The robot arms may clean by some other method as well e.g., by blowing air or vacuum suction. If a block is found to be dirty it needs to cleaned before painting.

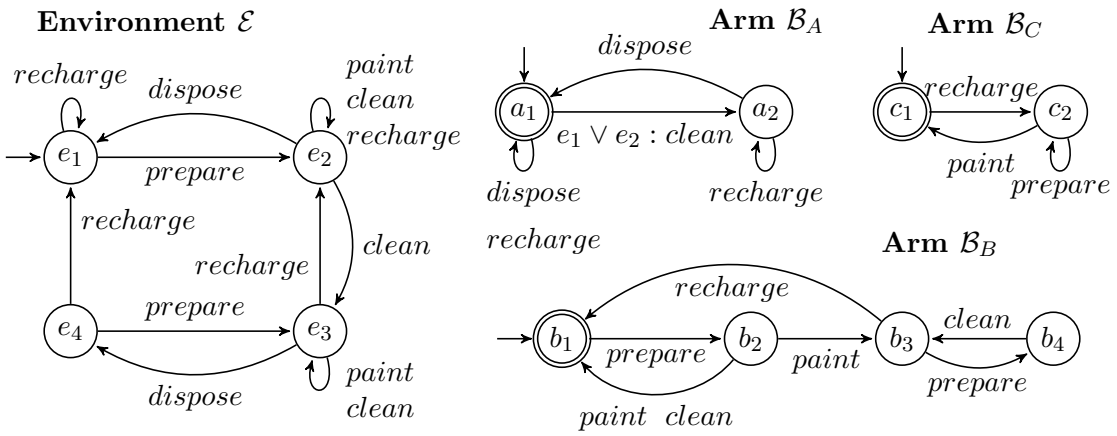


*Figure 2.1: The available system for the painting blocks domain.*

In this example, the environment (Figure 2.1)[1] has four states $e_1$, $e_2$, $e_3$ and $e_4$, with $e_1$ as the start state. The environment models the various restrictions and specifications of the painting blocks domain, e.g., the robot arms can *recharge* anytime. In particular, note that the start state of the environment $e_1$ only allows *recharge* and *prepare* actions. This forces that a block must be prepared before processing it. After preparing a block the environment moves to the state $e_2$, where the block can either be cleaned if it's dirty or it can be painted. Since after executing the *clean* action the water tank in environment can be empty the clean action is non-deterministic, i.e., if after cleaning a block the water tank goes empty, the environment evolves to the $e_3$ state, else if the tank still has water it remains in the $e_2$ state. In states $e_3$ and $e_4$ the water tanks are empty whereas in $e_1$ and $e_2$ they are not. If a robot arm is using air for cleaning it can still perform the clean action in the $e_3$ state as water is not consumed for such an action.

### 2.1.2 Behaviors

A behavior models the program or logic for a device which has an applicable set of actions for each of its states. Behaviors present the choice of available actions, depending on their current state, to the controller. Furthermore, behaviors act in an environment and they can be tested for conditions for their applicability i.e., the behavior can have restrictions on the environment states in which it can perform an action, e.g., if a behavior uses water to clean the blocks, then, such an action can only be executed in environment states where the water tank is not empty. Such an action is said to have *guards* with respect to the environment

[1] All figures were taken from published paper [Sardina et al., 2008] with explicit permission from the authors

states.

Formally, a behavior over an environment $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$ is $\mathcal{B} = \langle B, b_0, \mathcal{G}, \sigma, F \rangle$, where:

- $B$ is a finite set of behavior states;
- $b_0 \in B$ is the initial state;
- $\mathcal{G}$ is the set of *guards*, that is a Boolean function $g : E \mapsto \langle true, false \rangle$ where $E$ is the set of environment states in $\mathcal{E}$;
- $\sigma \subseteq B \times \mathcal{G} \times \mathcal{A} \times B$ is the transition relation: $\langle b, g, a, b' \rangle \in \sigma$, or $b \xrightarrow{g,a} b'$ in $\mathcal{B}$, denotes that the action $a$ in environment state $e$ when $g(e) = true$ will cause the behavior to move from the state $b$ to its successor state $b'$.
- $F \subseteq B$ is the set of final states of the behavior in which the behaviour may stop executing.

Blocks-world domain has three robot arms as given behaviors (Figure 2.1). Arm $\mathcal{B}_A$ can *clean* and *dispose* blocks, arm $\mathcal{B}_B$ can *prepare*, *clean* and *paint* blocks, and the arm $\mathcal{B}_C$ can *paint* and *prepare* blocks. Arm $\mathcal{B}_A$'s *clean* action has guards on it as it uses water to clean. Therefore, the *clean* action can be performed only when the water tank is not empty in the environment, i.e., environment is in states $e_1$ or $e_2$. Arm $\mathcal{B}_B$ paints blocks consuming paint from its own tank. Since this is internal to arm $\mathcal{B}_B$'s behaviour, one cannot predict the robot arm's state after every paint action, i.e., the paint tank in the robot arm can either go empty, resulting in the arm being unable to paint until it's recharged, or if there is still paint left, that arm can continue to perform the paint action. Thus, the paint action is *non-deterministic* as evident from the state $b_2$ of arm $\mathcal{B}_B$. In contrast, arm $\mathcal{B}_B$ uses air to clean the blocks, hence there are no guards present on the clean action. Finally, Arm $\mathcal{B}_C$ is more conservative than arm $\mathcal{B}_B$, as it has a smaller tank capacity which holds enough paint for a single paint action; it has to recharge its tank after each paint action it performs. All the behaviors can stop executing when they reach their initial states.

### 2.1.3 Target

The target behaviour represents the desired functionality that is required to be achieved by suitably controlling the available behaviors. The target does not exist in reality, rather it is a form of a specification to which the available system should conform. The target behavior is required to be deterministic, with only one possible successor state for each action. In

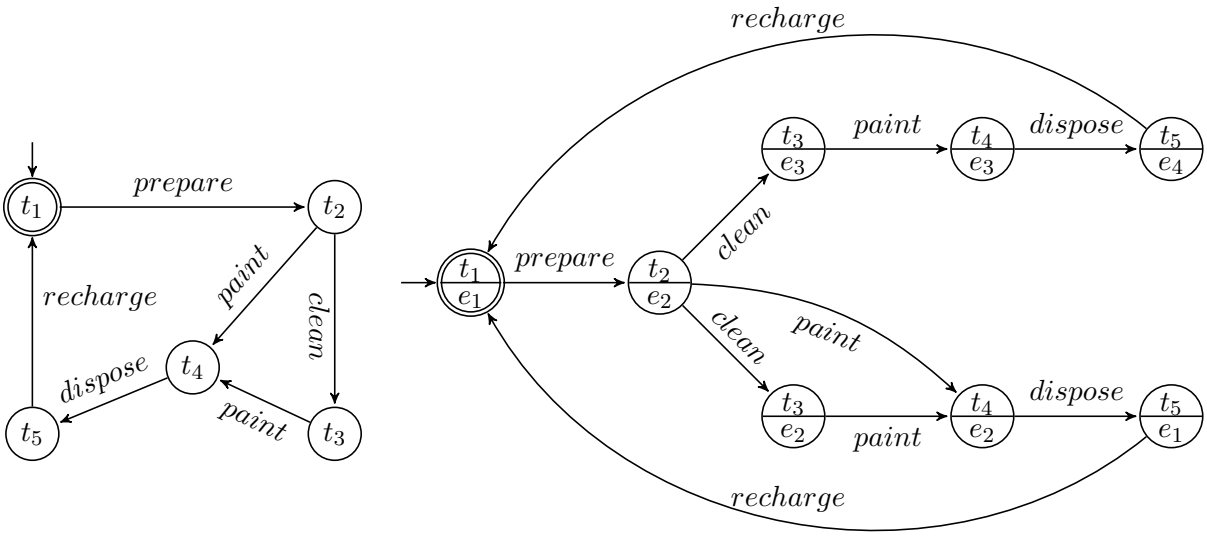


*Figure 2.2: The target behavior $\mathcal{B}_T$ (left) and the enacted target behavior.*

our example (Figure 2.2), it always starts with preparing a block and then depending on the block's condition proceeds to either clean and then paint it, if the block is dirty, or paint it without cleaning. Once the block is painted it is disposed and the tanks are recharged. Note, the target is conservative in its nature since it recharges in every cycle. The available behaviors need to enact the target by executing the actions one at a time, as specified by the target behaviour's specification.

## 2.2 The Composed System

All available behaviors act in a common environment, therefore, the actions that each behavior wants to execute must be supported by the environment's current state. Moreover, we assume that only one behavior can be active at a time, i.e., they execute actions one after another rather than concurrently.

Suppose that in the painting blocks domain the arm $\mathcal{B}_C$ is in state $c_2$ and the environment is in the state $e_4$. Although arm $\mathcal{B}_C$ can perform *paint* and *prepare* actions, only *prepare* can be executed, since the environment only supports the *prepare* action. Let us consider another example; imagine that arm $\mathcal{B}_A$ is in state $a_1$ and the environment is in state $e_3$. Though the

environment allows the *clean* action in its state $e_3$, arm $\mathcal{B}_A$ cannot perform the *clean* action due to the presence of its guards: it can perform the *clean* action only when the environment is in state $e_1$ or $e_2$.

In order to capture the overall capabilities of a behavior in the environment, we build the so called enacted behavior, i.e., what can a behaviour do while synchronously *enacting* in the common environment.

### 2.2.1 Enacted Behavior

A behavior's action causes both the behavior and the environment to evolve to their successor states. We can abstract this combined system of behavior and its environment like a finite transition system such that given a behavior $B = \langle B, b_0, \mathcal{G}, \sigma, F\rangle$ in an environment $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho\rangle$, the *enacted behavior* of $\mathcal{B}$ over environment $\mathcal{E}$ is defined as a tuple $\mathcal{T}_\mathcal{B} = \langle S, \mathcal{A}, s_0, \delta, Z\rangle$, where

- $S = B \times E$ is the finite set of $\mathcal{T}_\mathcal{B}$'s states, given a state $s = \langle b, e\rangle$, $b \in B$ and $e \in E$, we denote $b$ by *beh(s)* and $e$ by *env(s)*;
- $\mathcal{A}$ is the set of actions in $\mathcal{E}$;
- $s_0 = \langle b_0, e_0\rangle$ is the initial state of $\mathcal{T}_\mathcal{B}$, that is, both behavior and environment are in their initial states;
- $\delta \subseteq S \times A \times S$ is the enacted transition relation, where $\langle s, a, s'\rangle \in \delta$ or $s \xrightarrow{a} s'$ in $\mathcal{T}_\mathcal{B}$ iff (i) $env(s) \xrightarrow{a} env(s')$ in $\mathcal{E}$; and (ii) $beh(s) \xrightarrow{g,a} beh(s')$ in $\mathcal{B}$ with $g(env(s)) = true$ for some $g$ in $\mathcal{G}$;
- $Z \subseteq S$ is the set of final states of $\mathcal{T}_\mathcal{B}$, such that $Z = \{\langle b, e\rangle | b \in F, e \in E\}$ is the set of final states of $\mathcal{T}_\mathcal{B}$

Then, the enacted behavior is formally the *synchronous product* of the behavior and the environment. It represents all possible actions that can be performed by the behavior. Figure 2.3 shows the enacted behavior of arm $\mathcal{B}_C$. The initial state of the enacted behavior consists of the initial state of the arm $\mathcal{B}_C$ and the initial state of the environment, i.e., $\langle c_1, e_1\rangle$. Upon executing the *recharge* action, arm $\mathcal{B}_C$ moves to its state $c_2$ and the environment remains in the state $e_1$; therefore, the successor state for the enacted behavior for the action *recharge* from the initial state is $\langle c_2, e_1\rangle$. Observe, as pointed out before, that when the enacted behavior is in state $\langle c_2, e_1\rangle$ only the *prepare* action can be performed and not the paint action,

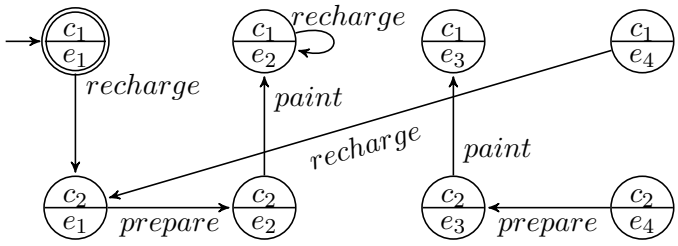


*Figure 2.3: Enacted behavior of Arm $\mathcal{T}_{\mathcal{B}_C}$.*

since the environment does not allow the *paint* action to be executed in the state $e_1$. The paint action can only be performed when both the behavior and the environment can execute them, i.e., in the state $\langle c_2, e_2 \rangle$ of the enacted behavior. Intuitively, the enacted behavior tells us what can be achieved if the behavior is placed in the environment.

Similarly, the *enacted target* (Figure 2.2) is constructed from the synchronous product of the target and the environment. Although the target behavior is deterministic, the enacted target is non-deterministic due to the non-determinism in the environment. Observe that the enacted target is non-deterministic for the *clean* action in the state $\langle t_2, e_2 \rangle$ of the enacted target. Intuitively, the enacted target tells us what the target can specify the available system to do.

Note that some of the states in the enacted behavior are not reachable from the initial states, .e.g, $\langle c_1, e_3 \rangle$ in Figure 2.3. However, since more than one behavior is present in the environment, those states can be reached by allowing other behaviors to operate.

### 2.2.2 The System

Since there is more than one behavior present with only one of them acting at a time, the environment evolves with respect to each behavior's actions. Thus, knowing the enacted behavior for all the behaviors independently is not enough to capture the whole system. Therefore, we extend the enacted behavior definition to abstract the whole system consisting of all the available behaviors and the common environment. The combined system is defined as $S = \langle \mathcal{B}_i, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$, where $\mathcal{E} = \langle A, E, e_0, \rho \rangle$ is the environment and $\mathcal{B}_i = \langle B_i, b_{i0}, G_i, \sigma_i, F_i \rangle$ are the available behaviors. The *enacted system behavior* of $\mathcal{S}$ is the tuple $\mathcal{T}_S = \langle S_S, A, \{1, \ldots, n\}, s_{S0}, \delta_S, F_S \rangle$, where

- $S_S = \mathcal{B}_1 \times \ldots \times \mathcal{B}_n \times E$ is the finite set of $\mathcal{T}_\mathcal{S}$'s states, given a state $s = \langle b_1, \ldots, b_n, e \rangle$,

we denote $b_i$ by $beh_i(s_S)$ , for $i \in \{1, \ldots, n\}$, and $e_i$ by $env(s_S)$;

- $A$ is the set of actions in $\mathcal{E}$;
- $s_{S0} \in S_S$ the initial state of $\mathcal{T}_\mathcal{S}$, with $beh_i(s_{S0}) = b_{i0}$, for $i \in \{1, \ldots, n\}$, and $env(s_{S0}) = e_0$, is $\mathcal{T}_\mathcal{S}$'s initial state ;
- $\delta_S \subseteq S \times A \times \{1, \ldots, n\} \times S$ is the enacted transition relation, where $\langle s_S, a, k, s'_S \rangle \in \delta_S$ or $s_S \xrightarrow{a,k} s'_S$ in $\mathcal{T}_\mathcal{S}$ iff
  - $env(s_S) \xrightarrow{a} env(s'_S)$ in $\mathcal{E}$;
  - $beh_k(s_S) \xrightarrow{g,a} beh_k(s'_S)$ in $\mathcal{B}_k$ with $g(env(s_S)) = true$ for some $g$ in $\mathcal{G}$; and
  - $beh_i(s_S) = beh_i(s'_S)$, for $i \in \{1, \ldots, n\} \backslash \{k\}$;
- $F_S \subseteq S_S$ the set of final states of $\mathcal{T}_\mathcal{S}$, with $beh_i(s_{Sf}) = b_{if}$, for $i \in \{1, \ldots, n\}$ such that $s_{Sf}$ is the final state of behavior $\mathcal{B}_i$ .

Technically, the enacted system behavior is the *synchronous product* of the environment with the *asynchronous product* of the available behaviors. The presence of the index $k$ in the formal definition adds the restriction that only one behavior executes an action at a time. That is, upon executing an action only the environment and that particular behavior evolve synchronously to their successor states (the rest of the behaviors remain in their current states).

As an example let us study some of the interesting properties of the enacted system taking the painting blocks domain (see Figure 2.4). The initial state $\langle a_1, b_1, c_1, e_1 \rangle$ for the enacted behavior has all the available behaviors and the environment in their initial states. The environment (see Figure 2.1) only supports the *recharge* and *prepare* actions from its initial state. From the collection of available behaviors (Figure 2.1), arms $\mathcal{B}_A$, $\mathcal{B}_B$ and $\mathcal{B}_C$ can execute the *recharge*, *prepare* and *recharge* actions from their initial states respectively. As a result, the enacted system can evolve to different states based on which behaviors execute. For example, if arm $\mathcal{B}_B$ performs the *prepare* action, the enacted system evolves to the state $\langle a_1, b_2, c_1, e_2 \rangle$; whereas if arm $\mathcal{B}_C$ performs the *recharge* action the system evolves to the state $\langle a_1, b_1, c_2, e_1 \rangle$. Notice that during every transition between the enacted system state only the performing behavior and the environment evolve, the rest of the behaviors remain stationary. From state $\langle a_1, b_2, c_1, e_2 \rangle$ both the actions *clean* and *paint* are non-deterministic, i.e., on performing the action there can be more than one possible enacted state to which the system can evolve. However, the sources of their non-determinism is different, in that the paint action is non-deterministic in the enacted system due to the non-determinism from

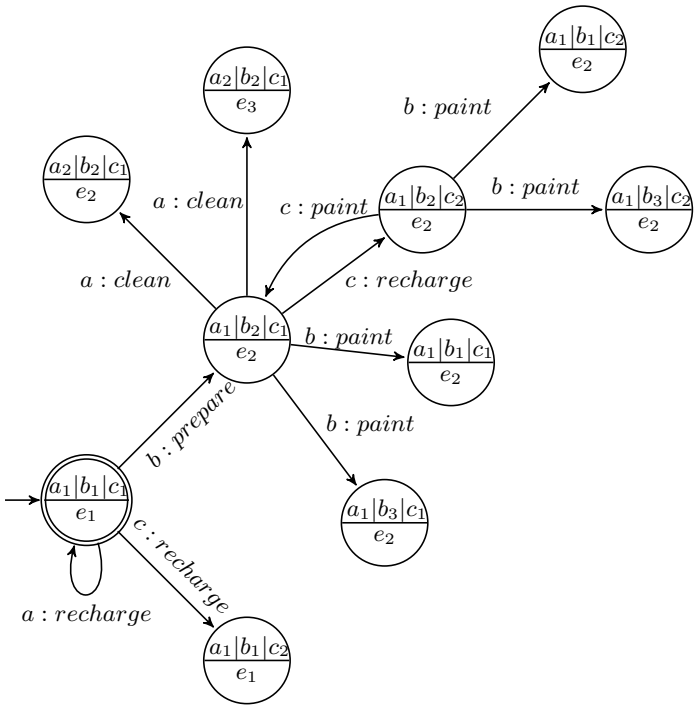


*Figure 2.4: Enacted System for the painting blocks example (Partial).*

the behavior of arm $\mathcal{B}_B$, whereas the clean action is reflected as non-deterministic due to the non-determinism present in the environment. The non-determinism available is compounded in the enacted system due to the presence of it in both environment and the behaviours. Suppose that the environment was non-deterministic with respect to the *paint* action as well, then, there would have been four different states (instead of two at the moment) that the enacted system could evolve as a result of the *paint* action. Furthermore, when the system is in the state $\langle a_1, b_2, c_2, e_2 \rangle$ and the paint action is to executed, there are two applicable behaviors, arms $\mathcal{B}_B$ and $\mathcal{B}_C$ that can accomplish the request. The enacted system can evolve to different states depending on which behavior is activated.

Since the enacted system should always be in a state from where it is able to do actions as per the target's specification, the choice of behavior in such cases depends on what actions the target can perform in future. Suppose that we choose a behavior whose action results in an enacted state which has behaviors in states such that none can perform the next action of the target, then, such a choice of behaviour would result in breaking of the solution.

## 2.3 Controllers and Compositions

What we have done above is describe all the components that build the behavior composition framework. We now state formally what the behavior composition problem amounts to, that is, what exactly is the task to be performed. We do so by defining the notions of *controllers* and that of *composition*.

**Behavior Controllers** A *controller* is a system component able to activate, stop, and resume any of the available behaviors, and to instruct them to execute an action among those allowed in their current state, also taking into account the state that the environment is in. We assume the controller has *full observability* of the available behaviors and the environment, that is, it can keep track (at runtime) of their current states.

In order to formally define controllers, let us first introduce some preliminary notions. Let $\mathcal{T} = \langle S, \mathcal{A}, s_0, \delta \rangle$ be an enacted behavior of some (available or target) behavior $\mathcal{B}$ over an environment $\mathcal{E}$.

A *trace* for $\mathcal{T}$ is a possibly infinite sequence $\tau = s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \cdots$, such that:

- $s^0 = s_0$;
- $s^j \xrightarrow{a^{j+1}} s^{j+1}$ in $\mathcal{T}_i$, for all $j > 0$.

A *history* is a finite prefix (ending with a state) $h = s^0 \xrightarrow{a^1} \cdots \xrightarrow{a^\ell} s^\ell$ of a trace. We denote $s^\ell$ by $last(h)$, and $\ell$ by $length(h)$ or, equivalently, $|h|$. Observe that finite traces are also histories, so function *length* is also defined over them. We extend *length* (or $|\cdot|$) as $length(\tau) = \infty$, if $\tau$ is infinite.

The notions of trace and history extend immediately to enacted system behaviors, by adding index $k$: system traces have the form $s^0 \xrightarrow{a^1,k^1} s^1 \xrightarrow{a^2,k^2} \cdots$, and system histories have the form $s^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell$. Functions *length* ($|\cdot|$) and *last* are extended in the obvious way.

Consider the enacted behavior for the painting blocks example (Figure 2.4). A possible history of length 3 starting from the initial state $\langle a_1, b_1, c_1, e_1 \rangle$ is $\langle a_1, b_1, c_1, e_1 \rangle \xrightarrow{prepare,2} \langle a_1, b_2, c_1, e_2 \rangle \xrightarrow{recharge,3} \langle a_1, b_2, c_2, e_2 \rangle \xrightarrow{paint,2} \langle a_1, b_1, c_2, e_2 \rangle$.

At this point we have all the machinery to define what controllers are. Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system and $\mathcal{T}_\mathcal{S}$ be its respective enacted behavior. Let $\mathcal{H}$ be the set of all $\mathcal{T}_\mathcal{S}$ histories. A *controller* for (system) $\mathcal{S}$ is a total function $P : \mathcal{H} \times \mathcal{A} \mapsto \{1, \ldots, n, u\}$. Given an enacted

system behavior history (or simply a *system history*) $h \in \mathcal{H}$ and an action $a \in \mathcal{A}$ to be performed, $P(h, a)$ is interpreted as the index of the available behavior (i.e., $\mathcal{B}_{P(h,a)}$) that action $a$ is delegated to. Special value $u$ ("undefined") is introduced for technical convenience, so as to make $P$ a total function defined also over irrelevant histories or actions that no behavior can perform after the system has traversed history $h$.

**Problem statement** The behavior composition problem amounts to synthesizing, for a given system $\mathcal{S}$, a controller that *realizes* a desired target behavior. Let us formally define what this means.

We start by defining the notion of *realization*. Let $\mathcal{S} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\}$ be a system, $\mathcal{B}_T$ a target behavior with its corresponding enacted behavior $\mathcal{T}_T$ and $P$ a controller for $\mathcal{S}$. Moreover, let $\mathcal{H}$ be the set of all system histories (i.e., histories of enacted system $\mathcal{T}_\mathcal{S}$) and consider a $\mathcal{T}_T$ (i.e., target behavior) trace $\tau = s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \ldots$. We say that $P$ *realizes* $\tau$ if:

1. for all enacted system histories $h \in \mathcal{H}_{\tau,P} \subseteq \mathcal{H}$ (see below), if $|h| < |\tau|$ then $P(h, a^{|h|+1}) = k$ and $last(h) \xrightarrow{a^{|h|+1},k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ for some $s'_\mathcal{S}$, where $\mathcal{H}_{\tau,P} = \bigcup_\ell \mathcal{H}^\ell_{\tau,P}$ is the set of $\mathcal{T}_\mathcal{S}$ *histories induced by* $P$ *and* $\tau$, inductively defined as follows:

   (a) $\mathcal{H}^0_{\tau,P} = \{s_{\mathcal{S}0}\}$;

   (b) $\mathcal{H}^{j+1}_{\tau,P}$ is the set of all $(j+1)$-length histories $h' = h \xrightarrow{a^{j+1},k^{j+1}} s^{j+1}_\mathcal{S}$ such that:

   - $h \in \mathcal{H}^j_{\tau,P}$;
   - $env(s^{j+1}_\mathcal{S}) = env(s^{j+1})$;
   - $k^{j+1} = P(a^{j+1}, h)$, that is, after history $h$, action $a^{j+1}$ is delegated to behavior $\mathcal{B}_{k^{j+1}}$;
   - $last(h) \xrightarrow{a^{j+1},k^{j+1}} s^{j+1}$ in $\mathcal{T}_\mathcal{S}$, that is, behavior $\mathcal{B}_{k^{j+1}}$ can actually perform action $a^{j+1}$ (taking also into account current environment state $env(last(h))$);

2. for each state $s^\ell$ occurring in $\tau$, if $beh(s^\ell) \in F_T$ (i.e., $beh(s^\ell)$ is final for $\mathcal{B}_T$) then all $\ell$-length histories $h \in \mathcal{H}^\ell_{\tau,P}$ are such that $last(h) \in F_\mathcal{S}$.

The above definition defines what constitutes a good controller. Intuitively, the set of histories $\mathcal{H}_{\tau,P}$ are the system histories that could be obtained as a result of the controller $P$ trying to realise the target trace $\tau$. A good controller should always be able to extend such histories (i.e. the ones in $\mathcal{H}_{\tau,P}$) by choosing a behavior such that the system can evolve legally without getting stuck.

The first condition tells us how to build the system histories. The system histories are built by starting from the start state of the enacted system (e.g. $\langle a_1, b_1, c_1, e_1 \rangle$) and then extending them in a stepwise manner. A history is extended by evolving the system such that the action is performed (as specified by the next action by target trace $\tau$) by the behavior chosen by the controller. For example, for the target trace $\langle t_1, e_1 \rangle \xrightarrow{prepare} \langle t_2, e_2 \rangle \xrightarrow{clean} \langle t_3, e_3 \rangle$, a good controller will take the history $\langle a_1, b_1, c_1, e_1 \rangle \xrightarrow{prepare} \langle a_1, b_2, c_1, e_1 \rangle$ and extend it by allocating the *clean* action to arm $\mathcal{B}_A$ so that the resulting history is $\langle a_1, b_1, c_1, e_1 \rangle \xrightarrow{prepare} \langle a_1, b_2, c_1, e_1 \rangle \xrightarrow{clean} \langle a_2, b_2, c_1, e_3 \rangle$.

Note that the environment is common to both the target and the available behaviors. Therefore, the environment's evolution remains the same, i.e., despite non-determinism in the environment, if the environment evolves to a state as a result of a target action, it will evolve to the same state as a result of executing the same action by a behavior.

The second condition states that while executing the sequence of actions specified by the target, if the target reaches a final state in a step then, all the behaviors should be in their final states during that step.

If a controller is able to guarantee all this, i.e., it can activate behaviors by delegating them actions so as to always *mimic* the target behavior, then the target is realizable by the available system. A controller $P$ realizes a target behavior $\mathcal{T}_T$ if it realizes all the traces of $\mathcal{T}_t$.

Formally, the problem can be stated as follows:

> Given a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and a deterministic *target* behavior $\mathcal{T}_T$ over $\mathcal{E}$, synthesize a controller $P$ that realizes $\mathcal{T}_T$.

All controllers that are a solution to this problem are called *compositions* of $\mathcal{T}_T$ on $\mathcal{E}$.

# Chapter 3

# Approaches to Behavior Composition

Various approaches have been suggested to solve the behavior composition problem in recent AI literature. These include techniques based on propositional dynamic logic satisfiability (PDL) [De Giacomo and Sardina, 2007], synthesis using linear temporal logic (LTL) [Sardina and De Giacomo, 2008] and alternating-time temporal logic (ATL) [Felli, 2008], simulation-based approach [Sardina et al., 2008] and search-based approach [Stroeder and Pagnucco, 2009].

We say a behavior composition problem has a solution if there exists a "controller" which can *realise* the target by scheduling the available behaviors. Propositional dynamic logic uses satisfiability checking to check the existence of such a controller. The behavior composition problem is translated to a set of PDL formulae, and the existence of a model which satisfies the formula is tested. The synthesis using ATL and LTL logic approach follows a game playing strategy. They translate each available behavior into a *game player*, and each decision of the controller to a *player* move. Then, the existence of a controller is confirmed if a game player can always make a move in the game.

The simulation based approach [Sardina et al., 2008] uses the formal concept of *simulation* [Henzinger et al., 1995; Tan and Cleaveland, 2001] to solve the composition problem. Using a regression-like approach, we test whether the enacted system is able to "mimic" the enacted target. We start by building the enacted system and the enacted target (see Section 2.2). After that, we *assume* that each state in the enacted system can *simulate* every state of the enacted state. Then we iteratively check for states for which the assumption is wrong.

On the other hand, the search-based approach [Stroeder and Pagnucco, 2009] starts with the initial enacted and target states, and then gradually builds the enacted system by expanding it as per the target specifications and verifying that one would never get "stuck".

In this chapter, we discuss the simulation based regression approach, including the formal definition of simulation. Following that, we briefly describe the search-based progression based approach.

## 3.1 Simulation-Based Regression Approach

One way to solve the composition problem is by applying the formal notion of *simulation* [Henzinger et al., 1995; Tan and Cleaveland, 2001]. The regression based approach by Sardina, Patrizi & De Giacomo [Sardina et al., 2008] uses *simulation* to find the solution to the behavior composition problem. Conceptually, a transition system simulates another system if it can mimic all its *moves*. Using this concept, a simulation is a *relation* between transition systems states which behave in the same way, i.e., one state simulates the other state.

Formally, given two transition systems $T_s = \langle \mathcal{A}, \mathcal{S}_s, s_{s0}, \delta_s \rangle$ and $T_t = \langle \mathcal{A}, \mathcal{S}_t, s_{t0}, \delta_t \rangle$, the simulation of $T_t$ by $T_s$ (i.e., $T_s$ simulates $T_t$) is a relation $R \subseteq \mathcal{S}_t \times \mathcal{S}_s$ satisfying the following for every tuple $\langle s_t, s_s \rangle \in R$ and $a \in \mathcal{A}$ :

- if $\langle s_t, a, s'_t \rangle \in \delta_t$ (i.e., $s_t \xrightarrow{a} s'_t$), then there is a state $s_s \in \mathcal{S}_s$ such that $\langle s_s, a, s'_s \rangle \in \delta_s$ (i.e., $s_s \xrightarrow{a} s'_s$) and $\langle s'_t, s'_s \rangle \in R$

In the rest of the thesis, we use the term *link* to depict a tuple in such a relation, i.e., if $\langle s_t, s_s \rangle \in R$, then we say that there is a *simulation link* between $s_t$ and $s_s$.

Notice that the definition of simulation is inductive in nature. With respect to the behavior composition problem, a state in the enacted system is in the simulation relation with respect to a state from the enacted target iff

1. it can do all the actions that the target state can do; and
2. all the successor states of that action continues to remain in the simulation relation.

When a transition system $T_1$ *simulates* another transition system $T_2$ implies that $T_1$ can do all what $T_2$ can. However, the opposite might not be true ($T_2$ may not be able to simulate $T_1$).

In order to explicitly include environment and non-determinism in the simulation definition, a variant *ND-simulation* [Sardina et al., 2008] is used. Let the system of available behaviors be $S = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and let $\mathcal{B}_t$ be the target behavior over environment $\mathcal{E}$. Let $\mathcal{T}_S = \langle S_S, A, \{1, \ldots, n\}, s_{S0}, \delta_S \rangle$ and $T_t = \langle S_t, A, s_{t0}, \delta_t \rangle$ be the enacted system and enacted target behavior in the environment $\mathcal{E}$. An *ND-simulation relation* of $\mathcal{T}_t$ by $\mathcal{T}_\mathcal{S}$ is a relation $R \subseteq S_t \times S_S$, such that $\langle S_t, S_s \rangle \in R$ implies:

- $env(s_t) = env(S_S)$;
- for all $a \in A$, there exists a $k \in \{1, \ldots, n\}$ such that for all transitions $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$:
  - there exists a transition $s_S \xrightarrow{a,k} s'_S$ in $\mathcal{T}_S$ with $env(s'_S) = env(s'_t)$; and
  - for all transitions $s_S \xrightarrow{a,k} s'_S$ in $\mathcal{T}_\mathcal{S}$ with $env(s'_S) = env(s'_T)$, we have $\langle s'_t, s'_S \rangle \in R$

The definition adds the constraint that both the given system and the target system are in the same environment state, and that despite the non-determinism the successor states still remain in the simulation relation.

Computing a *ND-simulation* relation between the given system and the target system constitutes the solution to a behavior composition problem.

Algorithm 1 calculates this *ND-simulation* relation. It starts with the assumption that each enacted system state simulates every enacted target state, i.e., it starts with the relation $\mathcal{R}$ having all the tuples $S_t \times S_\mathcal{S}$. Then it iteratively removes the tuples from $\mathcal{R}$ which violate the simulation definition. Informally, a tuple $\langle s_t, s_\mathcal{S} \rangle$ "breaks" the simulation definition if either of the following happen:

a) There exists an action that the enacted target state $s_t$ can do but the enacted system state $s_\mathcal{S}$ cannot. That is, none of the available behaviors can execute this action from their corresponding state in $s_\mathcal{S}$.

b) State $s_t$ is a final state but $s_\mathcal{S}$ is not.

c) The enacted system state $s_\mathcal{S}$ can do all the actions that the enacted target state $s_t$ can, but successor states of $s_\mathcal{S}$ are not in simulation with respect to the successor states of $s_t$.

The algorithm terminates when no more tuples can be removed from $\mathcal{R}$. The final resulting $\mathcal{R}$ is the *ND-simulation relation*. In [Sardina et al., 2008]; an effective procedure was shown to generate a controller with the simulation relation.

**Algorithm 1** $NDS(\mathcal{T}_t, \mathcal{T}_\mathcal{S})$ – Largest ND-Simulation

1: $\mathcal{R} := S_t \times S_\mathcal{S} \setminus \{\langle s_t, s_\mathcal{S}\rangle \mid env(s_t) \neq env(s_\mathcal{S})\}$

2: **repeat**

3: $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$, where $\mathcal{C}$ is the set of $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{R}$ such that either:

1. $s_t \in F_T$ and $s_\mathcal{S} \notin F_S$; or
2. there exists a transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$ such that for each $k$ either:
   (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$; or
   (b) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$ but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}$.

4: **until** $(\mathcal{C} = \emptyset)$

5: **return** $\mathcal{R}$

Let us take an example from the painting blocks domain (Figure 2.4) to elaborate over these notions. Consider the enacted system state $\langle a_1, b_2, c_1, e_3\rangle$ with respect to the enacted target state $\langle t_3, e_3\rangle$, i.e., the tuple $\langle\langle t_3, e_3\rangle, \langle a_1, b_2, c_1, e_3\rangle\rangle$ in $\mathcal{R}$. As per the target specification, the next action to perform is *paint*. Amongst available behaviors, arm $\mathcal{B}_B$ and arm $\mathcal{B}_C$ have the ability to perform the *paint* action. Arm $\mathcal{B}_B$ can do *paint* in its state $b_2$ and arm $\mathcal{B}_C$ can do *paint* when it's in state $c_2$. In the enacted system state $\langle a_1, b_2, c_1, e_3\rangle$ arm $\mathcal{B}_B$ is in a state such that it can honor the target's request. Therefore, the first condition defined above is fulfilled. Furthermore, $\langle t_3, e_3\rangle$ is not the final state therefore, we do need to check the second condition.

Finally, after performing the paint action the enacted system state $\langle a_1, b_2, c_1, e_3\rangle$ can evolve to $\langle a_1, b_1, c_1, e_3\rangle$ or $\langle a_1, b_3, c_1, e_3\rangle$, this is because of non-determinism in arm $\mathcal{B}_B$. The enacted target will evolve to $\langle t_4, e_3\rangle$. The final condition states that the successor states of enacted system state, i.e.,$\langle a_1, b_1, c_1, e_3\rangle$ and $\langle a_1, b_3, c_1, e_3\rangle$, should be in simulation with respect to the successor states of the enacted target, i.e., $\langle t_4, e_3\rangle$. That is, the tuples $\langle\langle t_4, e_3\rangle, \langle a_1, b_1, c_1, e_3\rangle\rangle$ and $\langle\langle\langle t_4, e_3\rangle, a_1, b_3, c_1, e_3\rangle\rangle$ should be present in candidate relation $\mathcal{R}$. Assume that $\langle\langle t_4, e_3\rangle, \langle a_1, b_3, c_1, e_3\rangle\rangle$ is not present in the relation $\mathcal{R}$, then the tuple $\langle\langle t_3, e_3\rangle, \langle a_1, b_2, c_1, e_3\rangle\rangle$ will be removed from candidate relation $\mathcal{R}$, since the successor of $\langle a_1, b_2, c_1, e_3\rangle$ is no longer in simulation with respect to the successor of $\langle t_3, e_3\rangle$. Note, removal of a tuple from $\mathcal{R}$ can cause a chain-like effect causing removal of other tuples.

The algorithm terminates when there are no more tuples to be removed from $\mathcal{R}$. Then, if

the initial states are in the simulation relation, i.e., the tuple $\langle\langle t_1, e_1\rangle, \langle a_1, b_1, c_1, e_1\rangle\rangle$ is in $\mathcal{R}$, then the composition problem has a solution. That is, starting from the initial states of the environment and the behaviors, the target can be realised fully.

We would like to point out a few key strengths of this approach. Firstly, since the enacted system and the enacted target are calculated independently of each other, multiple targets can be tested with respect to the same enacted system. Once built the enacted system can be stored in memory, thereafter, existence of a controller with respect to each target can be computed. Secondly, the approach allows handling of unexpected situations such as behavior failures [Sardina et al., 2008] as, in contrast with PDL satisfiability and synthesis, it generates *all* possible controllers for a solvable composition problem.

## 3.2 Search-Based Progression Approach

The progression based approach by Stroeder & Pagnucco [Stroeder and Pagnucco, 2009] checks the existence of a controller to a composition problem using a technique similar to forward search. The core concept here is to gradually evolve the desirable system by following the target specification, and verifying that it is never possible to get "stuck".

Here, the concept of state (expansion state) is similar to decision nodes in search. An expansion state $es$ consists of the following:

- A state in the target behavior, let us call it $t_i$.
- A set of *obligations*, i.e., the set of actions and their resulting expansion states. The actions used here are as per, what the target can do from $t_i$.
- The predecessor state of system configuration.
- A behavior index and an action such that the behavior corresponding to the behavior index can honor the parent's obligation.

Note, each expansion state contains the predecessor system state and the next target state. Since, there is no predecessor to the initial states of the behaviors and environment, a special start expansion state is used. This start state has the initial target state and dummy null environment and behavior states. After that, expansion states are expanded based on the obligations set by the target specification.

If an expansion state cannot perform its obligations, then it is deleted. The deletion is performed by *marking* these "bad" states. A state is considered bad if:

- it cannot execute actions as per its parent's obligations;
- if the target state is final but the behavior states are not; or
- if the successors of the expansion state are bad.

Interestingly, the mentioned constraints are similar to the simulation definition. The expansion and the marking steps are performed alternatively until either no states can be expanded further or, the initial system states are marked. If the initial system states are marked, then there does not exist a controller which can realise the target using the available behaviors.

Figure 3.1 shows the initial few expansion nodes for the painting blocks example. The start node consists of the initial target state $t_0$ with environment and behavior states as "null". It then expands to the first expansion node (node 1 in the figure) having the initial states of the behaviors and the environment. As one can see, the node has the predecessor system configuration $\langle a_0, b_0, c_0, e_0 \rangle$ and the target state as $t_1$. The chosen behavior index is 1, i.e., arm $\mathcal{B}_B$ and the action is *prepare*. Therefore, in order for the target to reach state $t_1$, arm $\mathcal{B}_B$ needs to do *prepare* from the system state of $\langle a_0, b_0, c_0, e_0 \rangle$. After arm $\mathcal{B}_B$ does *prepare* the system goes to its configuration $\langle a_0, b_1, c_0, e_0 \rangle$. From state $t_1$, the target can do *paint* and *clean*, therefore, *paint* and *clean* are the obligations for the state with system configuration $\langle a_0, b_0, c_0, e_0 \rangle$ (node 1).

From state with configuration $\langle a_0, b_1, c_0, e_0 \rangle$, both arm $\mathcal{B}_B$ and arm $\mathcal{B}_A$ can perform the *clean* action, as evident from expansion states 2 and 3. However, the choice of the behavior to execute the action should be such that the resulting state can honor the obligations. If arm $\mathcal{B}_B$ is chosen to *clean*, the resulting state is such that the *paint* action cannot be performed by any of the behaviors. Therefore, state 2 is marked and subsequently deleted. In contrast, if arm $\mathcal{B}_A$ is chosen to perform *clean*, the resulting state is such that the obligation of *paint* can be honored. Hence, the algorithm proceeds to expand it further.

Suppose the state 3 is marked as bad. Then state 1 will also be marked as bad, since its successors for action *clean* have been marked. Since state 1 represents the initial state of the system, the algorithm will terminate reporting no solution.

Observe that the search-based technique involves generating the enacted system in a forward manner by allowing it to evolve to all possible system configurations as per the target. The core idea is to check that such a generation will never get stuck.

The progression approach is backed with a Java based proof-of-concept prototype implementation.

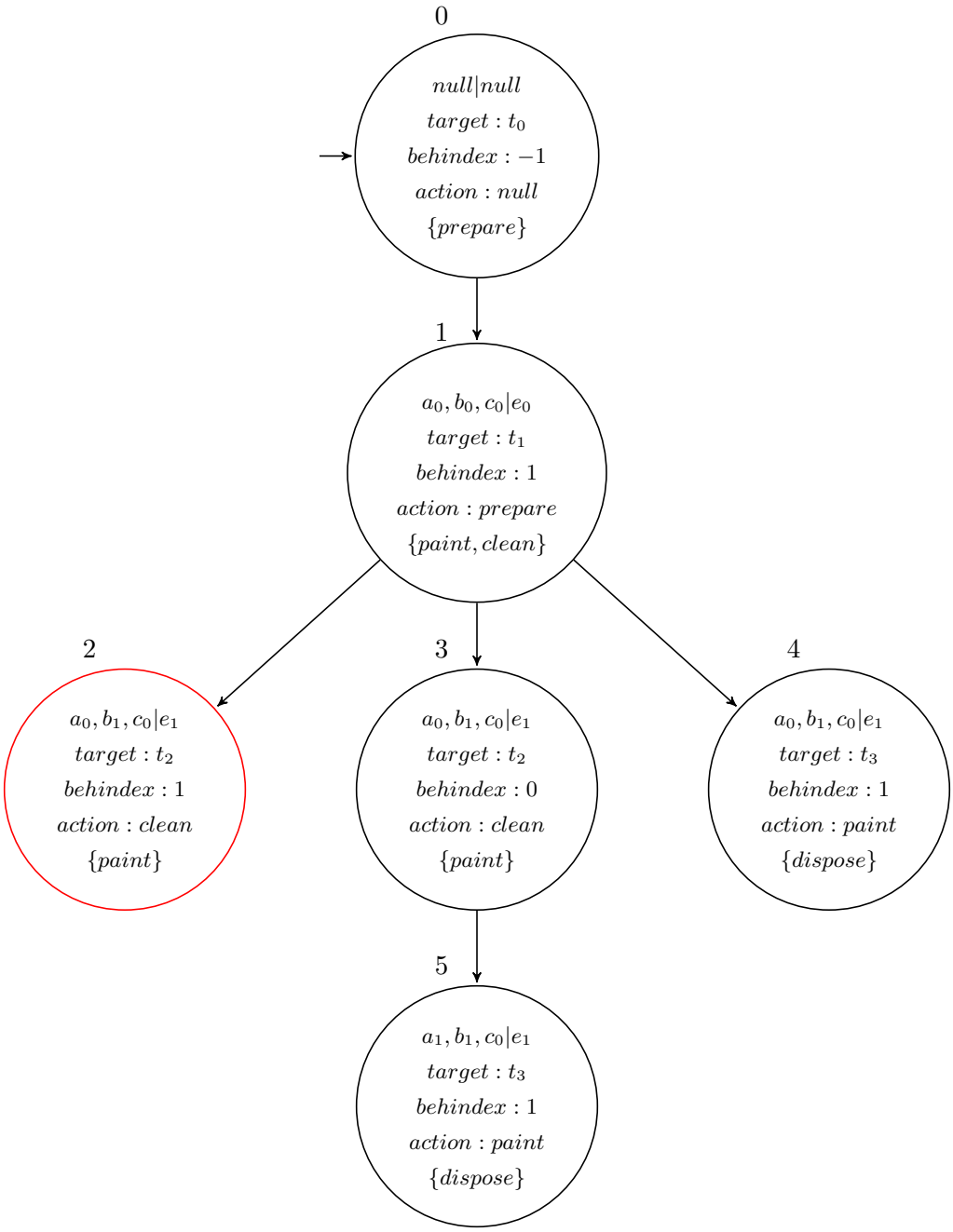


*Figure 3.1: Expansion States for progression approach for painting blocks example. In each state, the first line shows the behavior and environment state, the second line shows the target state, the third line shows the behavior index (starting from zero) chosen to perform the action (in the fourth line). The last line shows the obligations.*

# Chapter 4

# A Simulation-based Implementation

To our knowledge, the simulation-based regression approach has only been partially implemented. In particular, we are aware of two existing systems, namely Symfony [Almatelli Alessandro, 2009] and Opus [Balestra Concetta, 2009], that can solve the behavior composition problem [De Giacomo and Sardina, 2007; Patrizi, 2008].[1] Since both systems were developed in the context of web-service composition, where the task is to realize a target web-service by use of a set of existing web-services, neither of these systems account for the so-called environment (see Section 2.1.1). As a consequence of that, there is also no notion of "guards" in available behaviors (the feasibility of a transition in a behavior never depends on the state of the environment) and the concept of *enacted behavior* does not apply either. Rather, the existence of a solution is simply confirmed by checking that the initial state of the target web-service is in *simulation relation* with the initial state of the asynchronous product of the available web-services. In addition, both Symfony and Opus do not use data structures specially optimised for the behavior composition task, and hence, in some cases, it is difficult to judge whether poor performance is due to the intrinsic difficulty of the problem being solved.

In this chapter, we will describe our own fully-fledged implementation of the simulation-based technique described in Chapter 3. The aim is to have a high-quality and flexible implementation of the regression-like approach that can be used for experimentation and that allows for different extensions or optimisations. In fact, after describing the implementation for the basic technique, we will show how this can be easily extended to account for three different optimisations.

[1] We thank the authors of both systems and Prof. Giuseppe De Giacomo for allowing us to use their systems.

```
digraph service{
  b1 -> b2 [label = "c"]
  b2 -> b3 [label = "d"]
  b1 -> b3 [label = "c"]
  [initial = {b1}]
  [final = {b1,b2,b3}]
}
.
.
```

*Listing 4.1: Web service behavior description using DOT notation.*

```
digraph armA {
  a1 -> a1[label ="dispose"][legal={*}]
  a1 -> a1[label ="recharge"][legal={*}]
  a1 -> a2 [label="clean"][legal={e1,e2}]
  a2 -> a2 [label = "recharge"][legal={*}]
  a2 -> a1 [label = "dispose"][legal={*}]
  [initial = {a1}]
  [final = {a1}]
}
```

*Listing 4.2: Extended DOT description of the arm A from the painting blocks example.*

## 4.1 Allegro: The Core Composition System

The basic composition system, which we will refer to as Allegro,[2] implements the original regression-type Algorithm 1 (Chapter 3) from scratch. However, the system borrows and extends some of the elements from Symfony [Almatelli Alessandro, 2009] and the implementation of the search-based approach [Stroeder and Pagnucco, 2009]. Specifically, loading of the problem definition from text files is an extension of the file parsing functionality from Symfony, and the base data structures such as behavior and environment are borrowed from Stroeder & Pagnucco's implementation [Stroeder and Pagnucco, 2009].

### 4.1.1 File parsing extension

Symfony allows web-service composition using simulation-based regression approach. The inputs to the application are provided as text files, with one file per behavior, each are syntactically described using the DOT[3] notation (Listing 4.1). The JavaCC[4] compiler is used to parse the text files to allow creation of appropriate data structures in the memory. Symfony relies on the Jgraph[5] library for representing a finite transition system. In particular, it uses the directed multi-graph from the Jgraph package to store the various states and transitions of the behaviors. Symfony provides a good graphical interface for providing inputs to the system as well as for visualising the asynchronous product of the available behaviors, and the simulation relation.

[2] Allegro: composition or musical passage played with a brisk or rapid tempo.
[3] http://www.graphviz.org/doc/info/lang.html
[4] https://javacc.dev.java.net/
[5] http://jgrapht.sourceforge.net/

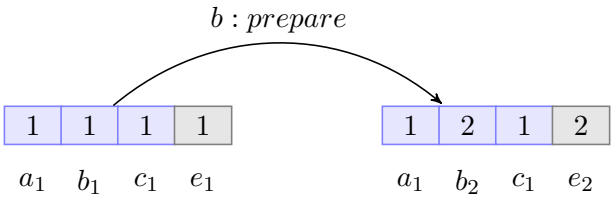


*Figure 4.1: Representation of states in enacted system.*

The grammar for describing the behaviors in Symfony was inadequate for our setting since guards for a behavior could not be expressed. Web service composition does not require an environment hence, the guards do not make sense in that setting. In order to be able to specify an environment and guards on behaviors with respect to it we present two extensions of the grammar. The changes done to the Symfony's grammar are as follows:

- Removal of the final state from the grammar for expressing environment;
- Ability to put guards in transitions for behaviors.

Compare the arm $\mathcal{B}_A$ (Listing 4.2), defined in the extended DOT syntax, with the web service example (Listing 4.1). Each behavior transition has specified legal environment states in which the transition can occur, e.g., the *clean* action can be done only when the environment is in states $e_1$ or $e_2$. The wildcard (*) implies that the transition can occur in all environment status or absence of any guard.

### 4.1.2 Enacted System

We build upon the base data structures from the search-based progression implementation by Stroeder & Pagnucco [Stroeder and Pagnucco, 2009]. Within a behavior, each state is identified with state name and a state number. An action causes a transition from a source (predecessor) state to a successor state. Actions are simply stored as strings, e.g., *clean*, *prepare*, *dispose*, etc. We build the enacted system based on these data structures.

Technically, an enacted system is also a transition system with each state composed of the behavior states and the environment state. We refer to a state in the enacted system as an *enacted state*. When a behavior executes an action in the enacted system, the action causes a transition from a source enacted state to a successor enacted state. The successor enacted state is such that, only the environment and the executing behavior change their respective states.

We represent each state of the enacted system as an array of integers (Figure 4.1) corresponding to the states of the behaviors and the environment. If the available system has $n$ behaviors, an enacted system state consists of an array of $n + 1$ integers, with the first $n$ integers corresponding to the current states of the behaviors and the last integer corresponding to the current state of the environment. This representation is analogous to the enacted system description (Figure 2.4).

In order to traverse through the enacted system efficiently a new data structure *StateLinks* is created. The primary purpose of *StateLinks* is to map each enacted state to its successor and the predecessor states. Conceptually, *StateLinks* consists of a map structure comprising of an action as the key and value being another map. This latter map has behavior index as the key and a list of enacted states as the value. Hence, given an action and a behavior index the *StateLinks* structure can retrieve successors and predecessors of an enacted state in (almost) constant time.

The enacted system then consists of pairs of enacted states with their *StateLinks*. Each enacted state has its associated *StateLinks* object. *StateLinks* allows a bi-directional navigation, i.e., it allows retrieval of both successors and predecessors from a given enacted state.

Since the actions are shared amongst behaviors and therefore, a given enacted state can be reached by performing the same action by different behaviors. Hence, while storing the predecessors of an enacted state, we group the predecessors with respect to actions and subsequently with respect to the behavior number depicting the behavior performing that action. The same grouping works even for storing the successors of an enacted state, since from an enacted state multiple actions are possible, with a single action being capable of being performed by multiple behaviors. Furthermore, due to non-determinism, an action performed by a behavior can lead to more than one enacted state.

For example, let us consider the enacted state $\langle a_1, b_2, c_2, e_2\rangle$ in Figure 2.4. From this enacted state, a *paint* action can be performed by two behaviors, arm $\mathcal{B}_B$ and arm $\mathcal{B}_C$. The *paint* action is non-deterministic for behavior arm $\mathcal{B}_B$, as a result, the enacted state can either evolve to $\langle a_1, b_1, c_2, e_2\rangle$ or $\langle a_1, b_3, c_2, e_2\rangle$. If arm $\mathcal{B}_C$ executes the *paint* action the system will evolve to $\langle a_1, b_2, c_1, e_2\rangle$. In our implementation of the enacted system, given the enacted state $\langle a_1, b_2, c_2, e_2\rangle$, action *paint* and, behavior arm $\mathcal{B}_B$ we can retrieve the successors $\langle a_1, b_1, c_2, e_2\rangle$ and $\langle a_1, b_3, c_2, e_2\rangle$ in (almost) constant time.

This kind of mapping enhances the ease with which the enacted system can be traversed as per the actions the target can request, as well as in, knowing the system capabilities in each state. In addition, the mapping provides an easy way of knowing if multiple behaviors are

available for executing an action, and in such conditions, if choosing a behavior breaks the solution then the solution with alternate behaviors can easily be checked.

For building the enacted system we start with the initial state of the system, e.g., $\langle a_1, b_1, c_1, e_1 \rangle$ for the painting blocks example, and evolve the system for all the applicable actions that the system can perform. We evolve the successor states and simultaneously update the *StateLinks* for each state. The complete enacted system is developed once there are no more new states to evolve.

Note, this approach builds only the *reachable* part of the enacted system from the initial state. In contrast, the approach suggested by the original description [Sardina et al., 2008] computes the reachable as well as the non-reachable parts of the enacted system. The computation of the non-reachable parts is helpful where behavior failure needs to be taken into account [Sardina et al., 2008]. Since we are only concerned with existence of a solution, therefore only reachable parts are computed. However, the implementation can be easily extended to support behavior failure.

### 4.1.3 Simulation Link

In order to store the simulation relation between the enacted system and the enacted target, we use a simple data structure called *SimulationLink* (Figure 4.2). It stores references to the *simulated* state of enacted target and the enacted system state which *simulates* it. The regression like algorithm operates on collection of such simulation links. For storing the simulation links we map them with respect to the enacted state. The map has the key as enacted state and the value as list of simulation links associated with the enacted state used as the key. Since the regression algorithm checks the simulation links between the enacted system state and the enacted target state, mapping with respect to enacted states provides a high iteration efficiency.

The mapping strategies, i.e., in the enacted system and the simulation links collection, allows easy checking of simulation links between the predecessor's and successor's of an enacted system stated and its corresponding simulated enacted target state. Although, some amount of filtering is still required to check simulation links between the predecessor's of enacted states, absence of such a mapping technique would imply searching in the complete simulation link collection.

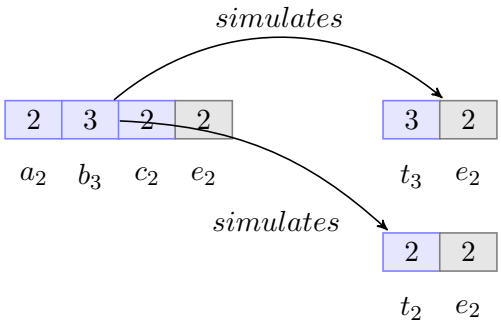


*Figure 4.2: Simulation link between an enacted system state and enacted target states.*

## 4.2 Optimisations

The original approach (Algorithm 1) follows a brute force kind of strategy in computing the simulation relation. We now present three optimisations over the original approach. The optimisations can be applied as standalone or in tandem with each other. Adding the suggested optimisations provides a more meaningful approach to simulation links checking and thereby reduces the computation effort in terms of the total links checked.

### 4.2.1 Initial optimisation

The original approach starts with the assumption that each enacted system state simulates every enacted target state. Then it removes the links iteratively which violate the simulation definition. More concretely, during each iteration it removes a link between a enacted system state $s_{\mathcal{S}}$ and an enacted target state $s_t$ if the link violates any of the following three checks:

- Can $s_{\mathcal{S}}$ execute all the actions that $s_t$ can request?
- If $s_t$ is final, is $s_{\mathcal{S}}$ final?
- Are the successors $s_{\mathcal{S}}$ and $s_t$ in simulation?

In the original approach, the first iteration of Algorithm 1 removes all the links that violate the first two checks as mentioned above. Therefore, checking for them again in every iteration does not provide any additional benefit, rather it does unnecessary work. Algorithm 2 optimises over the original approach in this aspect. Instead of starting with a links collection which has links from each enacted system state to every enacted target state, the optimised version only starts with a collection of links such that the enacted system state is capable of executing

**Algorithm 2** $NDS^1(\mathcal{T}_t, \mathcal{T}_\mathcal{S})$ – Initial step optimisation

1: $\mathcal{R} := S_t \times S_\mathcal{S} \setminus (\{\langle s_t, s_\mathcal{S}\rangle \mid env(s_t) \neq env(s_\mathcal{S})\} \cup \{\langle s_t, s_\mathcal{S}\rangle \mid s_t \xrightarrow{a} s'_t, \nexists i : s_\mathcal{S} \xrightarrow{a,i} s'_\mathcal{S}\} \cup \{\langle s_t, s_\mathcal{S}\rangle \mid s_t \in F_T \ , \ s_\mathcal{S} \notin F_S\})$

2: **repeat**

3: $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$, where $\mathcal{C}$ is the set of $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{R}$ such that there exists a transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$ such that for each $k$ there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ and $env(s'_t) = env(s'_\mathcal{S})$ but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}$.

4: **until** $(\mathcal{C} = \emptyset)$

5: **return** $\mathcal{R}$

all the actions that the target state can request and if the target state is final then the enacted system state is also final, i.e, the first two checks. Therefore, in each iteration the optimised algorithm only needs to remove links which violate the third check. This optimisation would not only save one iteration, but would also require less work to be done in each cycle. Theorem 1 shows that both the algorithms, original and initial step optimised, compute the same simulation set.

**Theorem 1.** *Given the same composition problem definition $NDS$ and $NDS^1$ compute the same simulation relation.*

*Proof.* Let $\mathcal{R}_i^1$ ($\mathcal{C}_i^1$ ) and $\mathcal{R}_i^2$ ($\mathcal{C}_i^2$ ) be the sets $\mathcal{R}$ ($\mathcal{C}$) in the $i^{th}$ iteration of Algorithm 1 and Algorithm 2, respectively, and $\mathcal{A} = \{\langle s_t, s_\mathcal{S}\rangle \mid s_t \xrightarrow{a} s'_t, \nexists i : s_\mathcal{S} \xrightarrow{a,i} s'_\mathcal{S}\} \cup \{\langle s_t, s_\mathcal{S}\rangle \mid s_t \in F_T \ , \ s_\mathcal{S} \notin F_S\}$, i.e., $\mathcal{R}_0^2 = \mathcal{R}_0^1 - \mathcal{A}$.

Next, we show that $\mathcal{R}_1^1 = \mathcal{R}_0^2$, that is the initial candidate set $\mathcal{R}$ of Algorithm 2 is equal to the candidate set $\mathcal{R}$ of Algorithm 1 after the first iteration. Then, $\mathcal{R}_1^1 = \mathcal{R}_0^1 - \mathcal{C}_1^1$. Let us next show that $\mathcal{C}_1^1 = \mathcal{A}$.

Suppose on the contrary, that $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{C}_1^1$ and $\langle s_t, s_\mathcal{S}\rangle \notin \mathcal{A}$. Therefore either :

1. $s_t \in F_T$, but $s_\mathcal{S} \notin F_S$. However, in that case, $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{A}$, thus reaching a contradiction.
2. there exists a $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$, such that either:
   (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$. However, in such a case, $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{A}$, thus reaching a contradiction; or
   (b) there exists $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$, but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}_0^1$. However, as $env(s'_t) = env(s'_\mathcal{S})$, then $\langle s'_t, s'_\mathcal{S}\rangle \in \mathcal{R}_0$, thus reaching a contradiction.

**Algorithm 3** $NDS^2(\mathcal{T}_t, \mathcal{T}_\mathcal{S})$ – Predecessor link checking optimisation

1: $\mathcal{R} := S_t \times S_\mathcal{S} \setminus \{\langle s_t, s_\mathcal{S}\rangle \mid env(s_t) \neq env(s_\mathcal{S})\}$

2: $\mathcal{P} := \mathcal{R}$

3: **repeat**

4: $\mathcal{C}$ is the set of $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{P}$ such that either:

1. $s_t \in F_T$ and $s_\mathcal{S} \notin F_S$; or
2. there exists $a \in \mathcal{A}$ for which for each $k$ there is a transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$ such that either:
   (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$; or
   (b) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$ but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}$.

5: $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$

6: $\mathcal{P} := \{\langle s'_t, s'_\mathcal{S}\rangle \mid \langle s'_t, s'_\mathcal{S}\rangle \in \mathcal{R}, \langle s_t, s_\mathcal{S}\rangle \in \mathcal{C}, s'_t \xrightarrow{a} s_t \in \mathcal{T}_t, s'_\mathcal{S} \xrightarrow{a} s_\mathcal{S} \in \mathcal{T}_\mathcal{S}\}$

7: **until** $(\mathcal{C} = \emptyset)$

8: **return** $\mathcal{R}$

Hence, $\mathcal{C}_1 = \mathcal{A}$ and $\mathcal{R}_1^1 = \mathcal{R}_0^1 - \mathcal{A}$. Therefore, $\mathcal{R}_1^1 = \mathcal{R}_0^2$.

It is easy to see then that when $\mathcal{R}_1^1 = \mathcal{R}_0^2$, then both the algorithms will compute the same set $\mathcal{R}$. Notice that conditions (1) and (2a) in Algorithm 1 apply only in the first iteration. □

Let us take an example from the painting blocks domain. Figure 4.3 shows a partial system with potential simulation links between the enacted system states and the enacted target states. Algorithm 1 starts with assumption that each enacted system state simulates every enacted target state, e.g., the state $\langle a_1, b_1, c_1, e_1\rangle$ has links to all enacted target states, i.e., links 1 to 6. In the next iteration it will remove links 2 to 6 since, the enacted state $\langle a_1, b_1, c_1, e_1\rangle$ cannot do actions which the enacted target states $\langle t_2, e_2\rangle$, $\langle t_2, e_2\rangle$, $\langle t_4, e_2\rangle$, $\langle t_3, e_3\rangle$, $\langle t_4, e_3\rangle$ can. In comparison, links 2 to 6 would not be present in the initial links set of Algorithm 2.

### 4.2.2 Predecessor link checking

As pointed out earlier, the simulation definition is inductive in nature. The removal of a link in an iteration might cause the removal of "predecessor" links in the next cycle. The "predecessor" link here means the link between the predecessor's of the given enacted system

state and the enacted target state, e.g., links 10 and 13 are the predecessor links for the link 12. The 2(b) check of Algorithm 1 removes a link if all the successors of a state are no longer in the simulation relation. Given this, we should only check the predecessor links of the links which are removed in the previous cycle. Re-checking of all links in every cycle is not necessary. Algorithm 3 defines the predecessor checking optimisation; being an optimisation the following proves that it is equivalent to Algorithm 1.

**Theorem 2.** *Given the same composition problem definition $NDS$ and $NDS^2$ compute the same simulation relation.*

*Proof.* Let $\mathcal{C}_k$ and $\mathcal{P}_{k-1}$ be the sets $\mathcal{C}$ and $\mathcal{P}$ at the end of Algorithm 1($NDS$) and Algorithm 3($NDS^2$) in the $k^{th}$ iteration, respectively. Then assume $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{C}_k$, that is, there is a simulation link removed between $s_t$ and $s_\mathcal{S}$ in the $k^{th}$ iteration by Algorithm 1. This implies that there exists a transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$ such that $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$, but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}_k$. This, in turn, implies that $\langle s'_t, s'_\mathcal{S}\rangle \in \mathcal{C}_{k-1}$, that is, $\langle s'_t, s'_\mathcal{S}\rangle$ was removed from $\mathcal{R}$ in the previous iteration. From here it is not difficult to see that $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{P}_{k-1}$. Hence, $\mathcal{C}_k \subseteq \mathcal{P}_{k-1}$ and $\mathcal{R}$ can be replaced by $\mathcal{P}_k$ in step 3 of Algorithm 1 without changing the meaning of $\mathcal{C}_k$.

□

Consider the link between $\langle a_1, b_3, c_2, e_2\rangle$ and $\langle t_4, e_2\rangle$, i.e., link 12 in Figure 4.3. Suppose, this link violates the simulation definition, and therefore, is removed in an iteration. In the next iteration, the original algorithm will still check all the remaining links, i.e., links 1 to 11. However, the only link that could possibly be affected by the removal of link 12 are link 10 and link 13, i.e., the link between the predecessors of $\langle a_1, b_3, c_2, e_2\rangle$ and $\langle t_4, e_2\rangle$. The predecessor link checking optimisation filters the links to be checked in the next iteration. For example, if link 12 is removed in an iteration, the optimised algorithm will only check links 10 and 13 in the next iteration.

Observe, the optimisation does not affect the number of iterations, but it checks fewer links in each iteration. Though in our implementation, retrieval of the predecessor states in the enacted system is done in almost constant time (see Section 4.1.2), the retrieval of predecessor links is not. This is because, in order to retrieve the predecessor links, e.g., link 12, we need to take the intersection of, links from the predecessor of the enacted system state, e.g., links 10, 13, and 8 and, the links to the predecessor of the enacted target state, e.g., 10, 9, 7, and 13

### 4.2.3 Initial state check

**Algorithm 4** $NDS^3(\mathcal{T}_t, \mathcal{T}_\mathcal{S})$ – Initial state check optimisation

1: $\mathcal{R} := S_t \times S_\mathcal{S} \setminus \{\langle s_t, s_\mathcal{S}\rangle \mid env(s_t) \neq env(s_\mathcal{S})\}$

2: **repeat**

3: $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$, where $\mathcal{C}$ is the set of $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{R}$ such that either:

1. $s_t \in F_T$ and $s_\mathcal{S} \notin F_S$; or
2. there exists a transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$ such that for each $k$ either:
   (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$; or
   (b) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$ but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}$.

4: **until** ($\mathcal{C} = \emptyset$ *or* $\langle s_{t0}, s_{\mathcal{S}0}\rangle \notin \mathcal{R}$)

5: **return** $\mathcal{R}$

A behavior composition problem has a solution if the initial system states have a simulation relation with respect to the initial target states. The original approach checks this only when the algorithm has finished executing, i.e., there are no more links to be removed. The initial state check optimisation (Algorithm 4) looks for the existence of the initial states in the simulation relation in each iteration. With the optimisation, the exit condition of the algorithm now includes two conditions i.e., if no more links are to be removed or if the initial states are no longer in the simulation relation.

Suppose in Figure 4.3, all the links from the initial state $\langle a_1, b_1, c_1, e_1\rangle$ (links 1 to 6) and the link between $\langle a_1, b_3, c_2, e_2\rangle$ and $\langle t_4, e_2\rangle$ (link 12) are removed in an iteration. Then, though the initial state has been removed from the simulation relation, the Algorithm 1 would still continue to check the remaining links. On the other hand, Algorithm 4 will stop (and return unsolvable) as soon as the initial state is removed from the simulation relation.

Algorithm 4 will stop executing as soon as it realises that a composition problem is unsolvable. This will save extra iterations in such cases. However, the number of iterations when there exists a solution is the same.

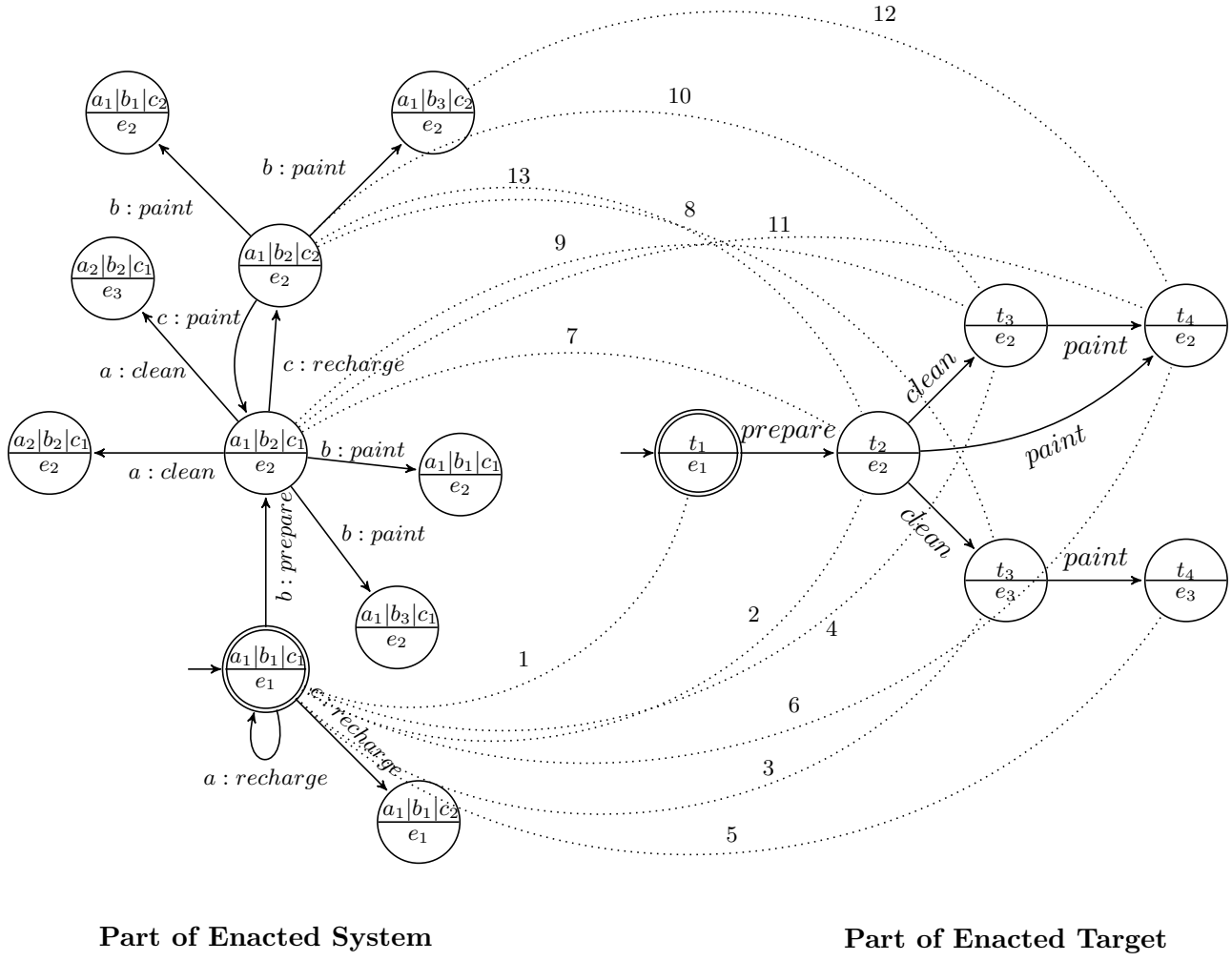


*Figure 4.3: Links between enacted system and enacted target for the painting blocks example (Partial).*

# Chapter 5

# Framework for behavior composition experiments

We design a framework which allows to empirically evaluate different approaches for solving the behavior composition problem. In our setting, we use the framework to evaluate the different optimisations for the simulation-based regression approach.

Until now there have been no known benchmarks designed for the behavior composition problem. The framework itself consists of a micro-benchmarking system, called Japex, integrated with a collection of generic methods to allow the creation of complex composition problems from a library of known problems. Since, creating meaningful problems for this domain takes a vast amount of time and it is hard to get solvable problems by randomly creating behaviors, we suggest a series of modular variations which can be applied to known problems. The main aim of the variations is to increase the complexities of the problem in a particular dimension, thus, allowing us to study the evolution of the problem with respect to its elements.

First, we start with a description of Japex; the benchmarking tool for Java programs, followed by details of the modular variations which can be applied to any existing problem. We also present an approach to create new problems based on the simple patterns with defined parameters.

## 5.1 Japex: A benchmarking toolkit

In order to benchmark an algorithm in Java extra precautions are required. Simply running the algorithm a few times and averaging the time may not lead to accurate results. The

recent versions of the Java virtual machine work on interpreting as well as optimising the frequently used byte code [Paleczny et al., 2001]. The Java HotSpot virtual machine works in a dual mode; interpreted and compiled. It interprets the code, profiles it and then compiles the frequently used code pieces. In addition, it can switch from interpreted to compiled code if required. The classes are loaded by the virtual machine in a lazy-loading fashion,[1] i.e., it loads a class only when it is required and not at program start-up. In addition, the Java garbage collector may become active during the program execution which further complicates the problem of benchmarking. Since most of these activities happen at run-time one cannot guarantee the accuracy of the execution time of an algorithm by an overall time measure.

Japex[2] is an open source tool for micro-benchmarking in Java which takes care of the garbage collection and lazy class loading concerns. It does this by measuring the garbage collection, class loading, and compilation time, and subtracting them from the overall execution time. In order to achieve this, the whole benchmarking process goes through four phases, namely, initialize, prepare, warmup, run, and terminate. In the initialize phase, the driver class is instantiated. During the prepare phase any data required can be loaded from a file or database. In the warmup phase Japex runs each algorithm to be tested for a specified time. Code warmup ensures that all classes have been loaded before benchmarking and Java virtual machine has reached its steady state execution stage. Lastly, in the terminate phase, any data or output can be saved. Japex runs the algorithm for a specified time and measures the number of iterations the algorithm does. It then divides the total time of the run phase by the number of iterations done to calculate the execution time.

In order to run a benchmarking test in Japex, a test suite is defined in a XML (see Appendix B.1) file which is given as input to the toolkit. A test suite contains a series of test problems, test drivers with specifications for runtime and warmup time respectively. A test driver (Listing 5.1) extends the base Japex driver class and provides methods corresponding to different phases of the benchmarking process. The benchmarking test problem specifications, e.g., the input files corresponding to the behaviors, target and the environment, are defined in another XML file which is passed to the drivers in the prepare phase. After which, the warmup and the run methods simply run the algorithms.

We create four different drivers, one each for the original regression approach (Allegro), the three optimisations (Allegro-OPT$^{1}$, Allegro-OPT$^{1+2}$, and Allegro-OPT$^{1+3}$), and one for the progression approach. Listing 5.1 shows the Japex driver for the original regression algorithm. In the prepare phase, we parse the input XML file defining the composition problem. During

[1] http://www.ibm.com/developerworks/java/library/j-benchmark1.html
[2] https://japex.dev.java.net

the warmup and the run phases we simply call the *simulate* method. The *simulate* method creates a new instance of the Allegro implementation, passes the required problem components, and runs the algorithm by calling its *compute* method.

```
public class RegressionDriver extends JapexDriverBase
{
        private ArrayList<Behaviour> available;
        private TransitionSystem environment;
        private Behaviour target;
        public void prepare(TestCase arg0) {
                String fileurl = arg0.getParam("inputfile");
                DriverHelper helper = new DriverHelper(fileurl);
                available = helper.getAvailableBehaviours();
                environment = helper.getEnvironment();
                target = helper.getTarget();
        }
        public void run(TestCase arg0) {
                simulate();
        }
        public void warmup(TestCase arg0) {
                simulate();
        }
        private void simulate() {
                BehaviourSimulatorMorpheus simulator = new
                     BehaviourSimulatorMorpheus(available, environment, target);
                simulator.compute();
        }
}
```

*Listing 5.1: Japex driver for regression algorithm*

## 5.2 Automatic generation of test cases

For creating test problems we use the already existing problems and manipulate their various components. We do this by creating modular methods and approaches which can be applied to any existing problem. This strategy allows creation of more complex problems from a library of known problems. All these variations are specified in the test problem XML file (see Listing 5.2). The test system applies the variations at runtime. This removes the need to create the newer problems by hand, rather we just need to input the base problem specification and the variations we wish to apply on it. Here we discuss the four ways we used to create new problems and how each of these affected the problem at hand.

| Number of arm $\mathcal{B}_C$ | Enacted system states |
|---|---|
| 1 | 48 |
| 2 | 96 |
| 3 | 192 |
| 4 | 384 |
| 5 | 768 |

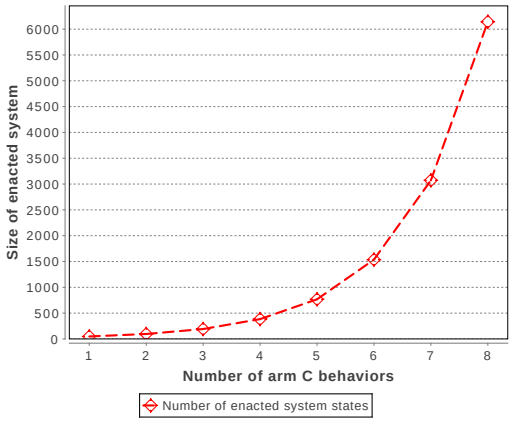


*Table 5.1: Number of enacted system states with respect to the number of arm$\mathcal{B}_C$ behavior*

| Number of arm $\mathcal{B}_B$ | Enacted system states |
|---|---|
| 1 | 48 |
| 2 | 192 |
| 3 | 768 |
| 4 | 3072 |
| 5 | 12288 |

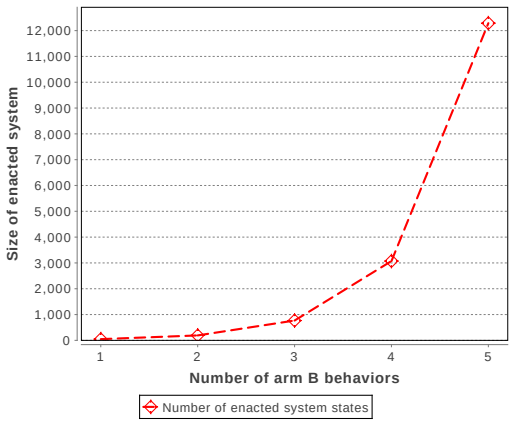


*Table 5.2: Number of enacted system states with respect to the number of arm $\mathcal{B}_B$ behavior*

### 5.2.1 Number of behaviors

In the first variation we increase the number of available behaviors in a behavior composition problem. The size of a behavior composition problem is directly related to the size of the enacted system and the enacted target. As discussed before (Section 4.1.2), an enacted system consists of the synchronous product of the environment with the asynchronous product of the available behaviors. By adding more behaviors to a problem, the size of the enacted system will increase, thus increasing the problem size. So, the first variation copies the existing behaviors several times. Listing 5.2 shows a sample XML configuration to define the variation wherein, the number of arm $\mathcal{B}_B$ used is 5.

For a given composition problem, if there exists a solution with the given behaviors, then increasing the number of any of the available behaviors will continue to have a solution. However, the size of the enacted system increases exponentially with increase in the number

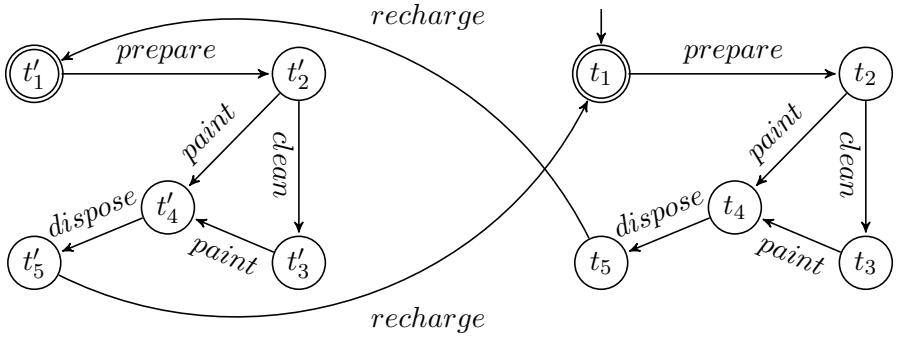


*Figure 5.1: Target Complexity for $\mathcal{B}_T$*

of behaviors.

Consider the effect of increasing the number of arm $\mathcal{B}_C$ behaviors on the number of enacted system states in the painting blocks example (Table 5.1). The number of enacted system states increase exponentially with respect to the number of arm $\mathcal{B}_C$ behaviors. With respect to arm $\mathcal{B}_B$, the increase in the number of enacted states is more. This is because arm $\mathcal{B}_B$ has 4 states whereas arm $\mathcal{B}_C$ as only 2. Note the size of the enacted system increases exponentially according to the number of states in the added behavior. In general, increasing the number of behaviors exponentially increases the problem complexity.

### 5.2.2 Target and Environment complexity

The other two components of the behavior composition problem are the environment and the target. In this section we show how to add complexity to these by making them perform extra *cycles* in their existing structure without changing the solution existence.

To amplify a transition system once, a replica of the existing structure is created. This replica is then connected with its original in a way so as to preserve the behavior's meaning. The replica is connected to the original by breaking and modifying transitions on a chosen state of the original transition system. This state is given as an input to the amplification method.

Formally, given a transition system $\mathcal{T} = \langle S, s_0, \mathcal{G}, \sigma, F \rangle$ and a state $\hat{s} \in S$, the amplified transition system is defined by $\mathcal{T}' = \langle S', s_0, \mathcal{G}, \sigma', F' \rangle$ such that:

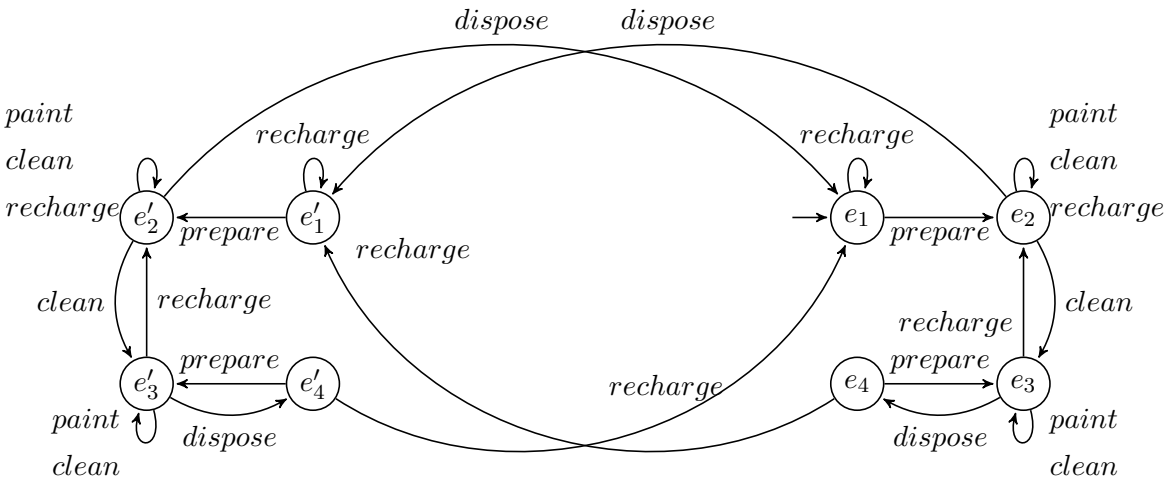


*Figure 5.2: Environment complexity*

1. $S' = S \cup \{s' \mid s \in S\}$.
2. $\sigma' = (\sigma - \{\langle s_1, g, a, \hat{s}\rangle \mid \langle s_1, g, a, \hat{s}\rangle \in \sigma\}) \cup \{\langle s'_1, g, a, s'_2\rangle \mid \langle s_1, g, a, s_2\rangle \in \sigma, s_2 \neq \hat{s}\} \cup \{\langle s, g, a, \hat{s}'\rangle \mid \langle s, g, a, \hat{s}\rangle \in \sigma\} \cup \{\langle s', g, a, \hat{s}\rangle \mid \langle s, g, a, \hat{s}\rangle \in \sigma\}$.
3. $F' = F \cup \{s' \mid s \in F\}$.

Figure 5.1 shows the effect of amplifying the painting blocks target by a factor of 1. Intuitively, in order to amplify a target given a state $b$ (e.g. state $t_1$ in the painting blocks target), we first delete the transitions that end in $b$ (e.g. $t_5 \overset{recharge}{\longrightarrow} t_1$). Then we create a copy of this transition system. Following that, we add a copy of the deleted transitions and modify them such that they now end in the copy of $b$ (i.e., state $b'$) corresponding state in the copied transition system (e.g. $t_5 \overset{recharge}{\longrightarrow} t'_1$). Finally, we add another copy of the deleted transitions and modify them such that their from states are the corresponding copies of the original from states (e.g. $t'_5 \overset{recharge}{\longrightarrow} t_1$).

Using this approach we can create new targets which do $n$ cycles, $n$ being the *amplification factor*. This amplification mechanism creates replicas of the original structure and allows transitioning to each copy. It preserves the original behavior and its capabilities and hence preserves the solution to the problem. More interestingly, if the given state is part of a cyclic structure, then the amplification allows the transition system to do multiple cycles of the same behavior. By allowing the target and the environment to do multiple cycles explicitly, we can add complexity into a given problem without changing the existence of the solution.

| Complexity | Enacted system | Enacted target |
|---|---|---|
| 1 | 48 | 8 |
| 2 | 96 | 16 |
| 3 | 144 | 24 |
| 4 | 192 | 32 |
| 5 | 240 | 40 |
| 6 | 288 | 48 |
| 7 | 336 | 56 |
| 8 | 384 | 64 |

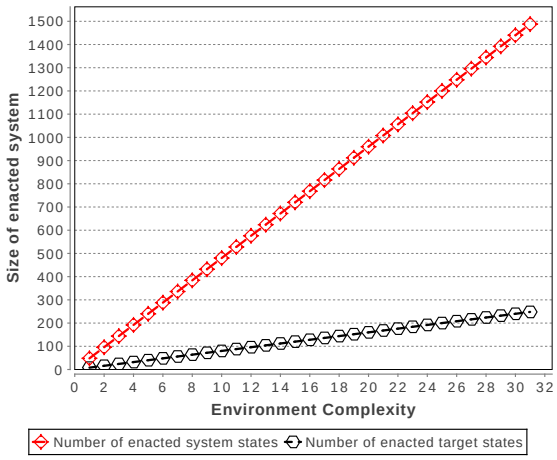


*Table 5.3: Number of enacted states with respect to environment complexity*

For example, in Figure 5.1, the actions that the target can do from state $t'_1$ are the same as from state $t_1$.

Similarly, the same method can be applied to the environment as well. Figure 5.2 shows the environment complexity applied to the painting blocks example. Here we amplify the environment using $e_1$ as the input state to the amplification method. There are two environment states $e_2$ and $e_4$ which transition to the state $e_1$. Thus the environment can not only do two cycles but can also shift from one copy to another, e.g., $e_2 \xrightarrow{dispose} e'_1$ transition shifts from the original to the copied transition system. Observe that, since both target and the available behaviors operate in the same environment, adding complexity to the environment affects both enacted system and enacted target.

The environment amplification creates copies of the existing states. As a result, the behavior and the target guards with respect to these may need to be updated. In fact, one needs to change the behaviors and the target to accommodate the copies of the environment states. For example, the transition $a_1 \xrightarrow{e_1 \vee e_2 : clean} a_2$ in arm $\mathcal{B}_A$ would need to be modified to $a_1 \xrightarrow{e_1 \vee e_2 \vee e'_1 \vee e'_2 : clean} a_2$ for the environment complexity shown in Figure 5.2.

Observe the effect of the environment complexity (Table 5.3) on the number of states in the enacted system and the enacted target. The number of states increase linearly in both the enacted system and the enacted target. The target complexity variation also causes the enacted target size to increase linearly.

The method can be easily extended to create problems *without a solution* by introducing a new action in the target. For example, in the last step of modification of the target (Figure 5.1), if we add the transition $t'_5 \xrightarrow{a} t_1$, this action cannot be performed by any of the behaviors.

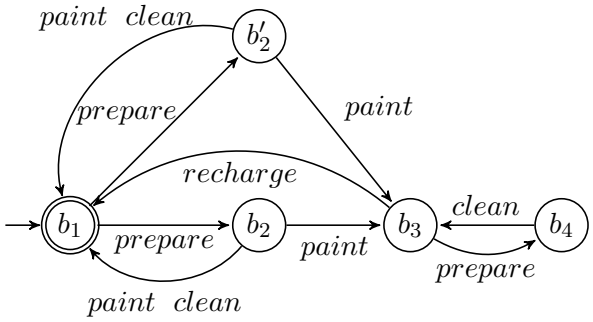


*Figure 5.3: Non-deterministic amplification applied to arm $\mathcal{B}_B$ of the painting blocks example.*

The environment also needs to be modified so that the target can perform $a$ in it. We do this by adding a self-looping transition to every state of the environment with the action $a$. This allows us to create problems for which we know no solution exists.

Listing 5.2 shows a sample XML configuration for the painting blocks example in which, the target is amplified 15 times and the environment is amplified 10 times.

### 5.2.3 Non-deterministic amplification

With the non-deterministic amplification modification we create multiple copies of a transition causing non-determinism. This amplification enhances the non-determinism in the behavior, and subsequently in the enacted system.

Formally, given a transition system $\mathcal{T} = \langle S, s_0, \mathcal{G}, \sigma, F \rangle$ and a transition $\langle s_1, g, a, s_2 \rangle \in \sigma$, the amplified transition system is defined by $\mathcal{T}' = \langle S', s_0, \mathcal{G}, \sigma', F' \rangle$ such that:

1. $S' = S \cup \{s_2' \mid s_2 \in S\}$;
2. $\sigma' = \sigma \cup \{\langle s_1, g, a, s_2' \rangle\} \cup \{\langle s_2', g', a', s \rangle \mid \langle s_2, g', a', s \rangle \in \sigma\}$;
3. $F' = F \cup \{s_2'\}$ if $s_2 \in F$, else $F' = F$.

Intuitively, the amplification method takes a transition (e.g. $b_1 \xrightarrow{prepare} b_2$ in arm $\mathcal{B}_B$) and creates copies of the successor state (e.g., $b_2$) in the transition (e.g., copied state $b_2'$). It then adds copies of the transition such that their successor state in each transition is a copied state (e.g., $b_1 \xrightarrow{prepare} b_2'$). Following that it creates copies of the transitions from the successor state (e.g., $b_2 \xrightarrow{paint} b_1$ and $b_2 \xrightarrow{paint} b_3$) and modifies them such that each transition is from a copied state (e.g., $b_2' \xrightarrow{paint} b_1$ and $b_2' \xrightarrow{paint} b_3$).

| ND Amplification | Enacted system states |
|---|---|
| 0 | 48 |
| 1 | 60 |
| 2 | 72 |
| 3 | 84 |
| 4 | 96 |
| 5 | 108 |

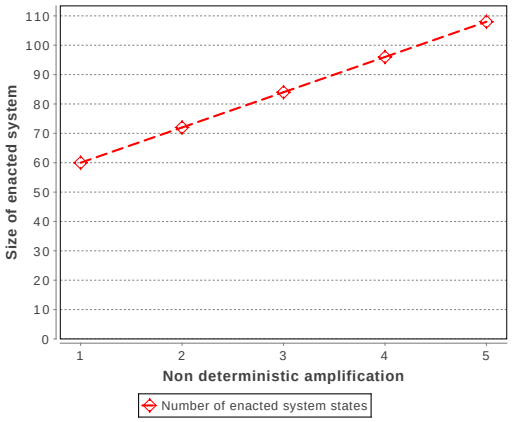


*Table 5.4: Number of enacted system states with respect to the ND Amplification of arm $\mathcal{B}_B$*

The non-deterministic amplification may create new sources of non-determinism in the behavior. For example, the action *prepare* in state $b_1$, which was earlier deterministic has now become non-deterministic.

We observe that the size of the enacted system increases linearly as a result of non-deterministic amplification (see Table 5.4). Since the non-deterministic amplification creates copies of an existing transition, the core behavior capabilities do not change, and therefore, do not affect the existence of a solution.

In Listing 5.2 the non-determinism in arm $\mathcal{B}_B$ is amplified by a factor of 2.

```
<tests>
<test>
        <environment amplification="10">environment.txt</environment>
        <behaviours>
                <behaviour times="1">armA.txt</behaviour>
                <behaviour times="5" nd="2">armB.txt</behaviour>
                <behaviour times="1">armC.txt</behaviour>
        </behaviours>
        <target amplification="15">target.txt</target>
</test>
</tests>
```

*Listing 5.2: Sample XML configuration for variations*

### 5.2.4 Simple Chains

Until now all the described methods took a known problem and applied different modifications to create new problems with added complexities. We now present a new generic method which

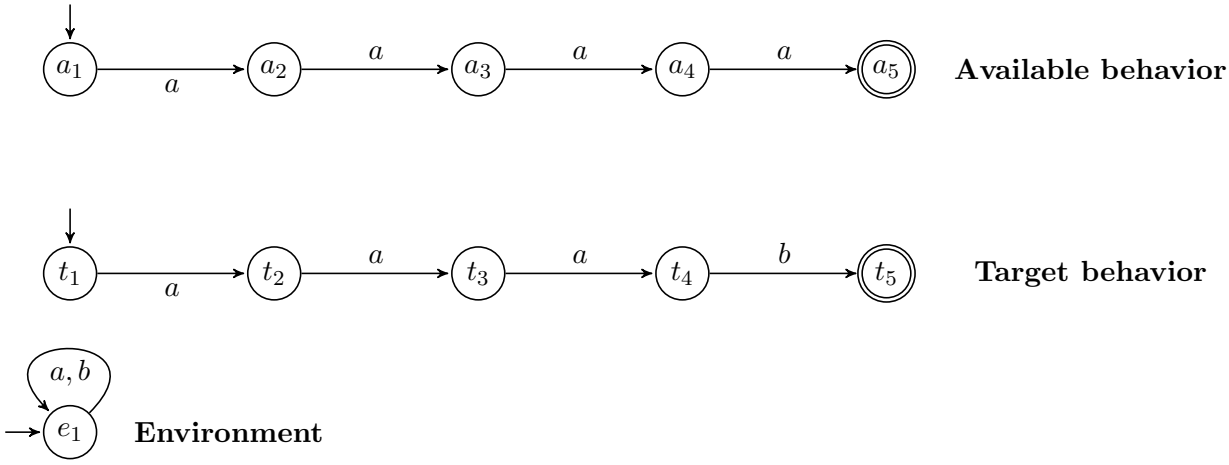


*Figure 5.4: Composition problem based on chain pattern with chain size of 5.*

creates a problem setting from scratch based on a chain pattern. This generic method allows creation of problems with given characteristics such as length of the chain, number of chains, etc.

We use this approach to specifically create problems for which we know no solution exists. In order to create such problems, the available behaviors can do all the actions that the target can request except the last action, i.e., the last action of the target is such that it cannot be done by any of the available behaviors. Figure 5.4 shows a sample problem based on the chain pattern with a chain size of 5.

# Chapter 6

# Experiments

We ran our implementation together with its optimisations on different composition problems. In particular, we used problems from two different domains described in the literature, namely, the painting blocks from Sardina et al. [2008] and, the rescue domain from Stroeder and Pagnucco [2009]. In addition, we create synthetic problems using the chain pattern described in Section 5.2.4. The framework described in Chapter 5 is used to benchmark the implementations.[1] Firstly, we apply modular variations to the basic problems, as discussed in Section 5.2. Lastly, we use a complex problem described from the web-services domain from Felli 2008, to compare our implementation with other regression based implementations. In some cases, we also test the search-based progression implementation. However, the search-based progression implementation is a proof of concept. We do not compare the efficiency of the approaches, but only observe the effect of the different modular variations.

We use the experiments to study two key aspects; a) effect of the variations on the problems and; b) analysis of optimisations for the regression approach.

In the results to follow, the unoptimised implementation is referred as Allegro, and the optimisations are named as:

- initial optimisation as Allegro-OPT$^1$ (which realises Algorithm 2);
- predecessor link checking optimisation as Allegro-OPT$^2$ (which realises Algorithm 3);
- initial state check as optimisation Allegro-OPT$^3$ (which realises Algorithm 4).

Since the optimisations can be applied in conjunction, Allegro-OPT$^{1+2}$ implies initial and

[1] All experiments were done on dual core 2Ghz 64 bit machines with 8GB of RAM.

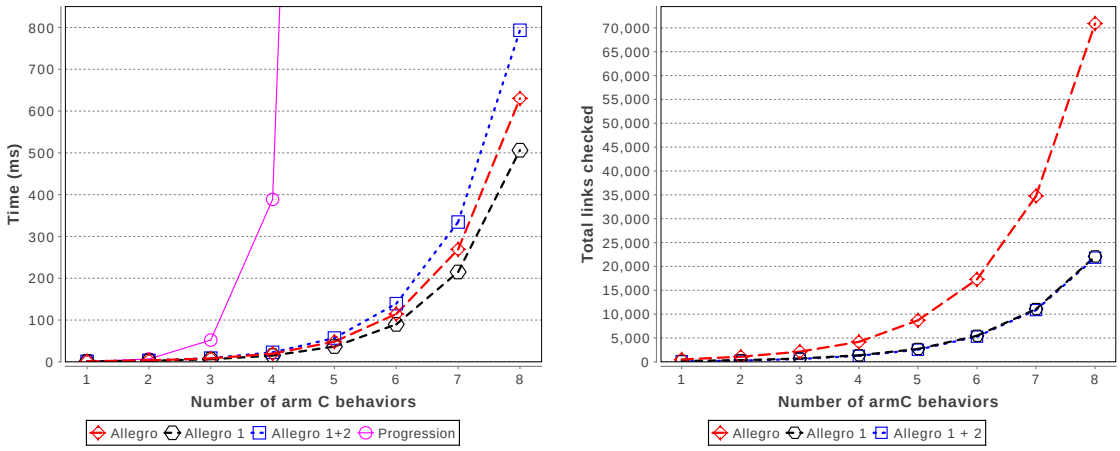


*Figure 6.1: Results for Number of behaviors variation for arm $\mathcal{B}_C$ of painting blocks domain*

predecessor link checking optimisations applied together and, Allegro-OPT$^{1+3}$ implies initial and initial state check optimisations applied together. In this chapter, we discuss the benchmarking results categorized as per the variations done on the problem sets.

## 6.1 Number of Behaviors

As discussed in Section 5.2, the increase in the number of behaviors causes the problem size to increase exponentially (see Table 5.1). This will indeed be apparent in our experiments.

In Figure 6.1, we can observe the performance of the different algorithms as we increase the number of arms of type $\mathcal{B}_C$ in the painting blocks domain from 1 to 8. Similarly, in Figure C.1, we depict the results for the rescue domain when we increase the number of diagnosis robots in the rescue domain. For comparing the different optimisations, we also plot the total number of links checked by each of the regression approaches.

First of all, the time taken by all the approaches increases exponentially, matching the exponential growth of the enacted systems. Second, the Allegro-OPT$^{1}$ approach takes less time than the unoptimised approach, due to less work done in each iteration, as explained in Section 4.2.1. This is verified by the evident reduction in the number of links checked. Finally, the Allegro-OPT$^{1+2}$ approach performs slower, even than unoptimised approach. The parent link checking optimisation involves filtering of parent links between the enacted state and the enacted target (Section 4.2.2). This filtering does not cause much difference in the total number of links checked as compared to Allegro-OPT$^{1}$, thus, the filtering is not beneficial in this scenario. Rather, the filtering causes an extra overhead. Therefore, the time taken by

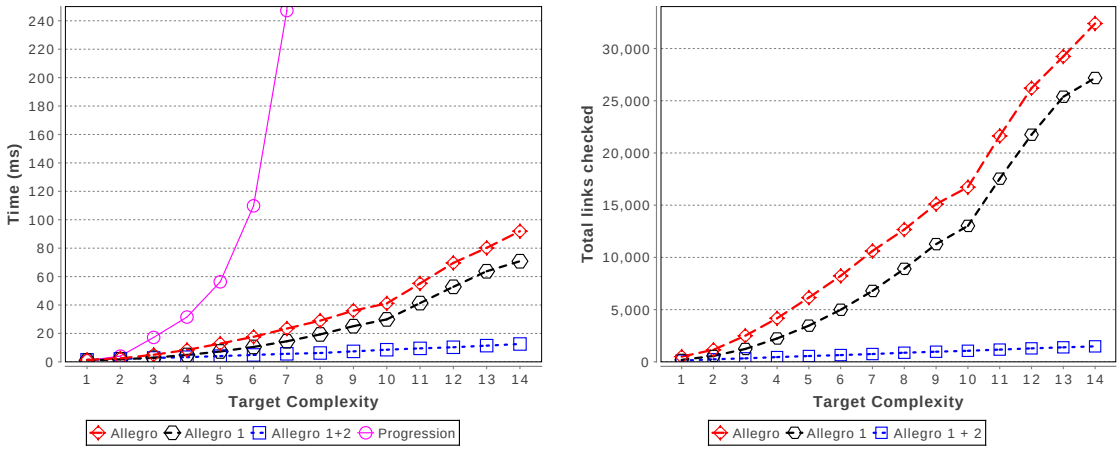


*Figure 6.2: Results for Target complexity variation for painting blocks domain*

Allegro-OPT$^{1+2}$ is more than the other regression implementations. In comparison, consider the rescue domain, the reduction of links checked increases with increase in the number of diagnosis robots (Figure C.1), as a result time taken by Allegro-OPT$^{1+2}$ approaches closer to the unoptimised approach. However, the link reduction is still not enough to provide a clear cut advantage.

## 6.2 Target and Environment Complexity

To study the effect of target and environment complexity in a composition problem, we amplify the target and the environment using the variations defined in section 5.2.2. The target in the painting blocks domain is amplified from 1 to 14, and the environment is amplified from 1 to 32.[2]

Figure 6.2 shows the results for the target complexity on the implementations for the painting blocks domain. All the simulation-based regression based implementations increase linearly, consistent with the linear increase in the size of the enacted as discussed in Section 5.2.2. In contrast, progression based approach does not increase linearly, we believe this is due to the use of recursion in the implementation. Note the progression approach increases linearly for the rescue domain, in fact it performs much better than the regression approach (Figure C.2).

The Allegro-OPT$^{1}$ algorithm performs better than the unoptimised implementation due to reduction in the total links checked. Observe, Allegro-OPT$^{1+2}$ performs faster compared to both, Allegro, and Allegro-OPT$^{1}$ implementations. In this setting the predecessor link filtering

[2]For simplicity we use non-guarded versions of the problems for environment complexity.

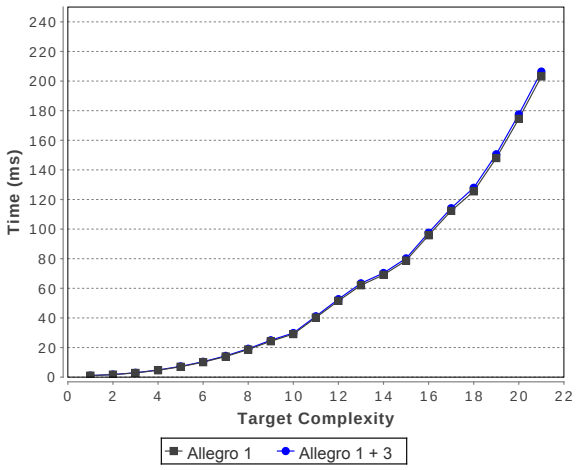


*Figure 6.3: Results for Initial state check for Target complexity variation in painting blocks domain.*

considerably reduces the total number of links checked. The reason is that target amplification causes the target to "cycle" multiple times, as long chains of the original target structure are formed. This results in an increase in the number of iterations with each target amplification, causing a decrease in the total number of links checked using the predecessor link checking optimisation, as apparent from Figure 6.2 (see below for discussion on chain structures). Amplification of the environment also shows similar results (Figure C.3 and C.4).

### 6.2.1 Initial state checking optimisation (**Allegro** Optimisation 3)

The initial state checking optimisation checks in every iteration whether the initial states still exist in the simulation relation. For unknown problems, one does not know if a solution exists or not. In cases where the composition problem has a solution, the Allegro-OPT$^{1+3}$ approach will involve an extra overhead. In contrast, in cases where there is not a solution for a problem, the Allegro-OPT$^{1+3}$ would be beneficial.

We test the Allegro-OPT$^{1}$ and Allegro-OPT$^{1+3}$ with the target amplification on the painting blocks example to measure the extra overhead. As evident from Figure 6.3, the difference in extra time taken by Allegro-OPT$^{1+3}$, as compared to Allegro-OPT$^{1}$, is minimal. The regular solution check causes minimal overhead in cases where a solution exists.

As discussed before, the target amplification creates "chains" of the original target structure. In order to test scenarios which do not have a solution, we introduce a new action in the target. Since none of the available behaviors have the ability to perform this action, we

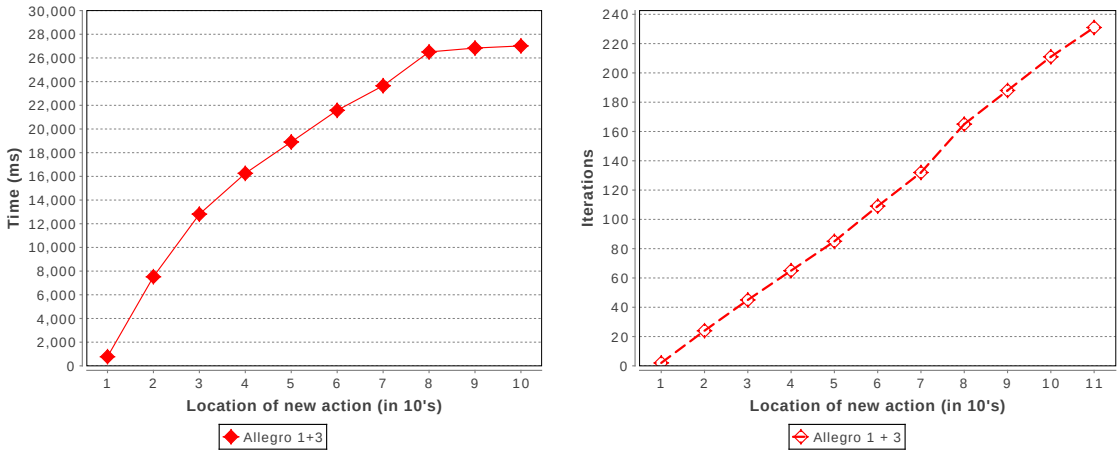


*Figure 6.4: Allegro-OPT$^{1+3}$ performance for problems without a solution in the painting blocks domain*

know for sure the target will not be realizable. We amplify the target by a factor of 200 and introduce the new action between the last and second last copies of the amplified target. Thereafter, we create test problems by inserting another new action after every 10 copies of the target, i.e., the first problem has the second new action after the $10^{th}$ copy, the second problem after the $20^{th}$ copy, and so on. Observe that the amplified target size remains the same, the only difference is in the location of the action that "breaks" the solution. What is important to see is that the size of the problem remains the same, but the time required is different depending on the location of the action that "breaks" the solution.

Figure 6.4 shows the results of Allegro-OPT$^{1+3}$ approach on these test problems. As the location of the new action goes farther from the initial states, more iterations are required to remove the initial state from the simulation relation. Using the Allegro-OPT$^{1+3}$ optimisation, as soon as the initial states are removed from the simulation relation, the algorithm terminates. Therefore, the farther the new action, the greater the number of iterations required before the algorithm terminates. Although, there may be more links which violate the simulation definition and could be removed in subsequent iterations, since the algorithm realises that a solution does not exist, it terminates. Intuitively, the Allegro-OPT$^{1+3}$ conveys what is the shortest distance we can go before realising that a solution does not exist. In comparison, without the regular solution check the algorithm does not finish until there are no more links to be removed, i.e., how far we can go before we realise a solution does not exist.

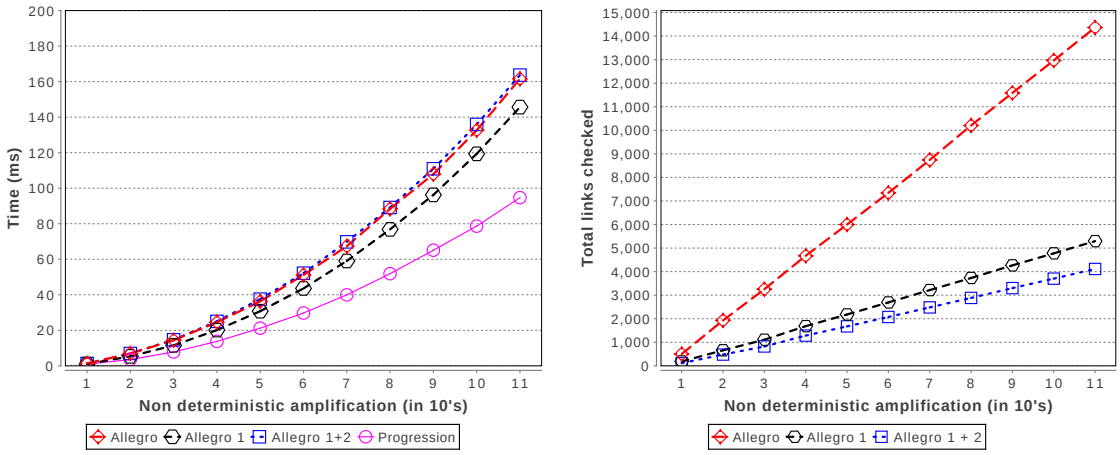


*Figure 6.5: Results for non-deterministic variation on arm $\mathcal{B}_B$ of the painting blocks domain.*

## 6.3 Non-deterministic Amplification

To study the effects of non-determinism in composition problems, we enhance the non-determinism in arm $\mathcal{B}_B$ of the painting blocks example from 0 to 100 in steps of 10, i.e., the first test is with no amplification, second test with amplification of 10, and so on. As discussed in Section 5.2.3, non-deterministic amplification increases the enacted system size linearly. Figure 6.5 shows a linear increase in time by all the approaches, conforming with our expectations. That is, Allegro-OPT$^1$ performs faster than Allegro, and Allegro-OPT$^{1+2}$ is the slowest of the regression approaches due to non-beneficial parent link filtering in this setting.

## 6.4 Chain Structures

We create synthetic problems based on the chain pattern as described in Section 5.2.4. The composition problems generated do not have a solution since the last action in the target chain can not be executed by the behavior. The composition problems have a single behavior. The variation done is in the length of the behavior and target chains. We increase the chain length, of both behavior and target, in multiples of 100. The first problem has a single behavior and target with chain length of 100, the second problem with chain length of 200, and so on.

Figure 6.6 shows results for the chains problem set. Firstly, Allegro-OPT$^1$ continues to perform faster than Allegro. Secondly, Allegro-OPT$^{1+2}$ performs substantially faster than Allegro and Allegro-OPT$^1$. This is due to the fact that predecessor link filtering in Allegro-OPT$^{1+2}$ decreases the total number of links checked by a large amount. Furthermore, since we have

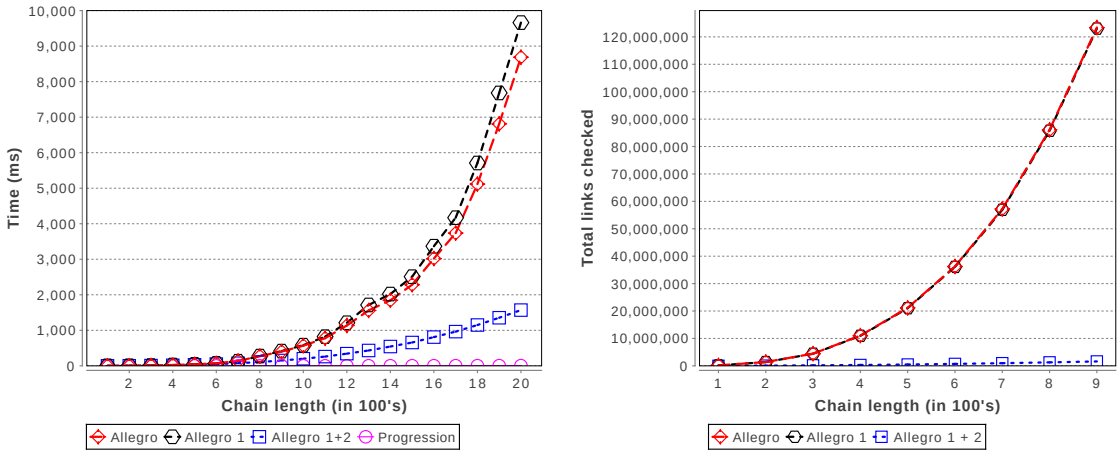


*Figure 6.6: Results for simple chains pattern.*

only a single behavior, the filtering is less expensive to perform. Specifically, for this example we have just one source state for every enacted state. We expect problems yielding behaviors similar to chain pattern to benefited from Allegro-OPT$^{1+2}$ optimisation.

If we visualise the enacted system as a graph, the lesser the in-degree as compared to the total number of nodes, the greater would be the benefit of Allegro-OPT$^{1+2}$ optimisation. The particular chain example above is an extreme case in which the in-degree is 1. Chain patterns in the enacted system can be observed in other cases, for example, in situations where behaviors have to act in an interleaved fashion, i.e., one behavior has to wait for another to act. In these cases, the algorithm will perform many iterations with each causing small changes in the candidate simulation relation. More concretely, as the ratio of the total iterations to total number of links reaches 1, the Allegro-OPT$^{2}$ optimisation benefit will increase.

## 6.5 Web-services Example

Lastly, we run a complex web-services example to test our implementation with other known regression-like implementations, namely Symfony and Opus, and the temporal logic verifier (TLV).

Figure A.2 depicts the complex web-services example taken from Felli [2008]. There are five available behaviors and a target, all with ten states each. Since our implementation requires on environment to be specified, we create a simple environment with a single state capable of doing all the actions in the domain.

Table 6.1 shows the time taken by various approaches. TLV is fastest of all the methods;

| Implementation | Time |
|---|---|
| TLV | 10 milliseconds |
| Allegro-OPT$^1$ | 23 seconds |
| Allegro | 51 seconds |
| Symfony | 50 minutes |
| Opus | 9+ hours |

*Table 6.1: Comparison of Allegro and Allegro-OPT$^1$ with other regression-like approaches*

it uses model checking techniques to check the existence of a controller by recasting the composition problems into game playing problems. Amongst the simulation-based regression implementations Allegro-OPT$^1$ is the most efficient by a large margin. The *StateLinks* (see section 4.1.2) data structure allows faster retrieval of successors and predecessors, reflected as increased speed in the algorithm. In addition, as done by Stroeder and Pagnucco [2009], we represent enacted states as an array of integers, thereby not creating complex state objects which themselves would slow the algorithm.

## 6.6 Summary

Based upon all the above experiment results, we draw the following conclusions:

1. Increasing the number of behaviors enhances the problem complexity exponentially. On the other hand, environment, target, and non-deterministic amplification increase the problem complexity linearly. This confirms the complexity results reported in [De Giacomo and Sardina, 2007; Sardina et al., 2008].

2. Initial optimisation (Allegro-OPT$^1$) and initial state check optimisation (Allegro-OPT$^3$) should be applied to all problems. Predecessor link checking optimisation (Allegro-OPT$^2$) should be applied in problems where the in-degree in the enacted system is considerably less than the number of enacted states (e.g., the chain pattern).

3. The Allegro system is an efficient implementation of the simulation-based technique. However, it is not clear this technique will outperform techniques based on game structures, as is the case with TLV.

# Chapter 7

# Conclusion

In this work we have provided the first complete implementation of the simulation-based regression approach for solving the behavior composition problem. We have also augmented the original technique with a set of three optimisations. We have shown that the initial optimisation will be beneficial in all situations, and therefore should always be applied. The initial state check optimisation provides benefits in problems which do not have a solution, with minimal overhead in cases where the solution exists. Since for unknown problems one does not know if a solution exists or not, this optimisation should be applied on all such problems. The predecessor link checking optimisation is clearly beneficial for chain-like patterns. There may be other patterns in which it might prove to be useful, these need to be empirically evaluated. In general, it is beneficial in problems where the in-degree is considerably less than the number of enacted states. Lastly, we developed a benchmarking framework for experimentation. In doing so, we developed modular variations which can be applied to existing problems to increase their complexities.

## 7.1 Future work

Currently, all the approaches synthesise a controller that can coordinate the available behaviours to realise the target behaviour. However, none of them provide any useful information if no composition solution exists; they just state that the composition problem is *unsolvable*. Since, in many domains, it would be conceivable that the desired module may not be realised with the available modules, it would be clearly advantageous to provide some useful information for non-solvable instances.

This can be done in at-least two ways, namely, *approximations* and *repairs*. The usefulness of approximation was already pointed out by Stroeder and Pagnucco [2009]. Rather than looking for a total (optimal) solution, the challenge is to approximate the target as much as possible: how much of the desired module can we cover with the available behaviours at hand? For example, in some cases one may be able to realise the target as long as some special actions are not requested by the target ever. When it comes to repairs, one is interested in obtaining information on how extra capability may help realise the target behaviour. For instance, it would be of great value to know if a (minimal) extension to an already existing available behaviour would suffice in order to make the problem solvable. Similarly, one may look for information on what kind of new behaviour module it would be enough to build (hopefully, less costly than building the target itself) so as to solve the problem .

Another interesting direction is the use of heuristic search to actually compute a solution. Current strategies employ blind search. The application of heuristics, as currently done in the field of automated planning by state-of-the-art planners, may drastically improve the search performance and help in finding solutions faster.

# Appendix A

# Examples

## A.1 Rescue Domain

Figure A.1 shows the urban search and rescue example [Stroeder and Pagnucco, 2009]. It uses a scenario in which robots are used to search and rescue humans in emergencies. It has three types of robots as available behaviors namely, scout robot, diagnosis robot and, rescue robot. Scout robot can *search* for humans and *report* their locations. Rescue robots *transport* the injured humans to a safe place. Diagnosis robots can *diagnose* injured people and can asses if they need *special* transportation. It can also help people to reach a safe location

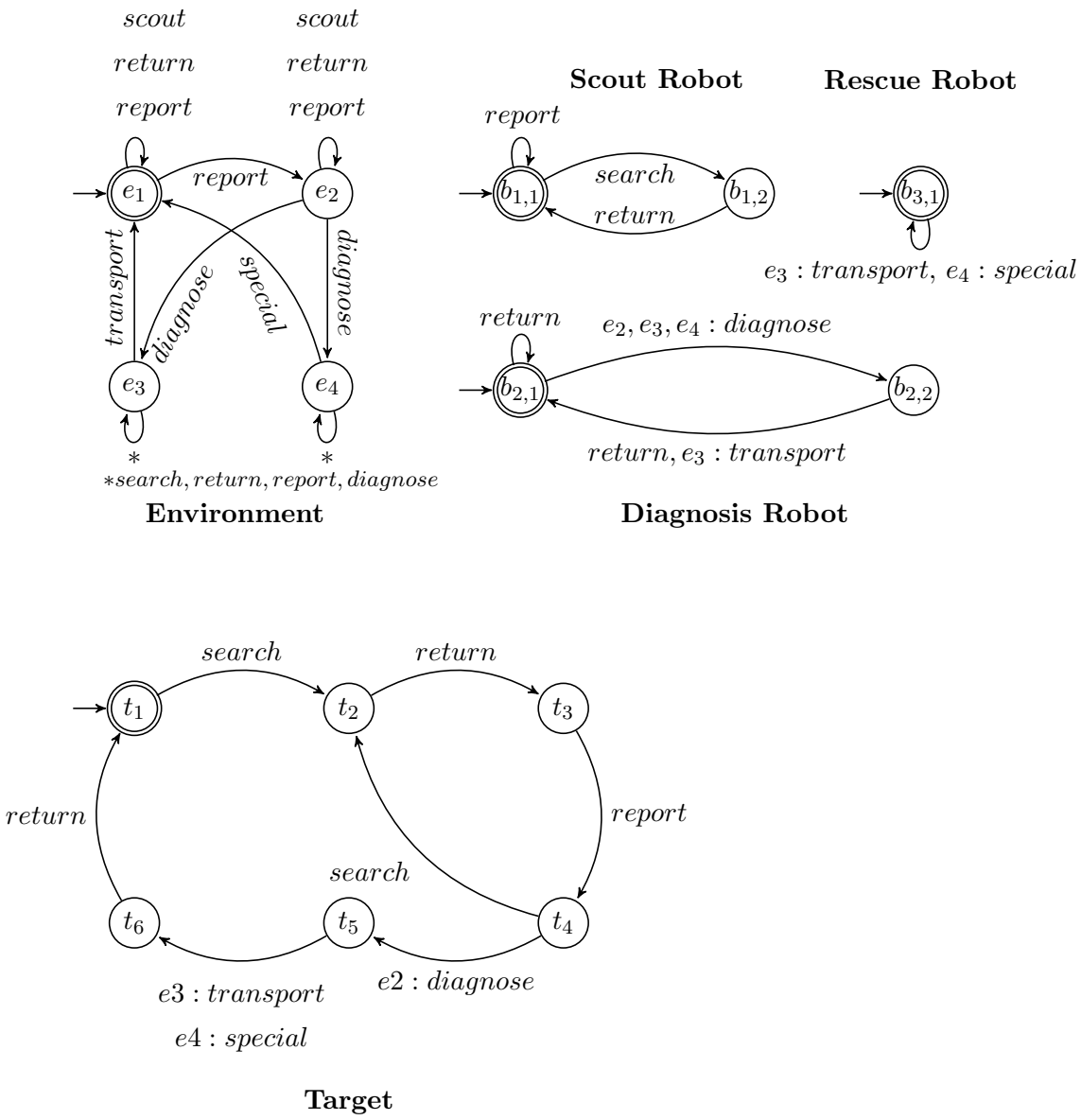


*Figure A.1: Rescue Domain Example [Stroeder and Pagnucco, 2009].*

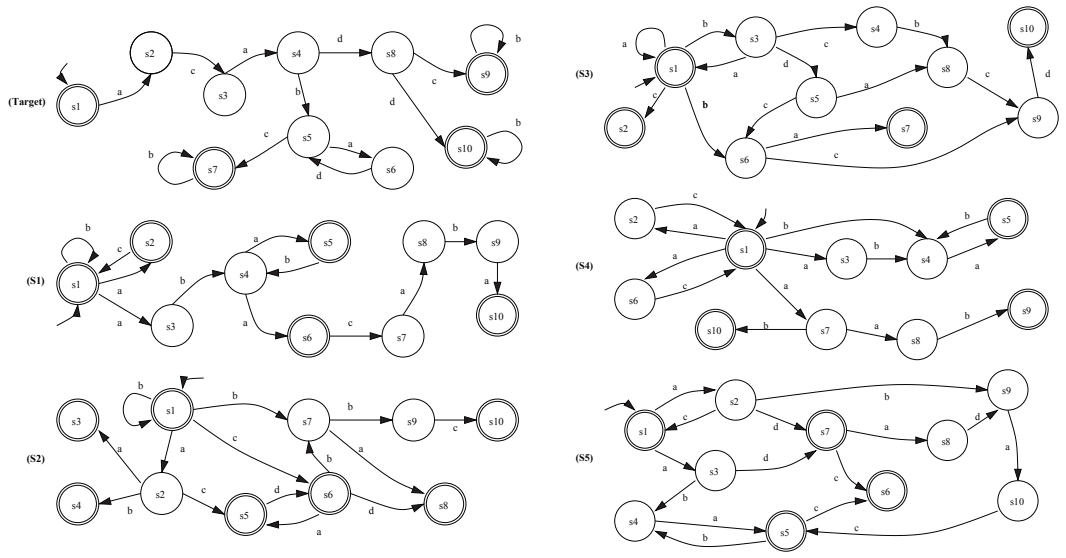


*Figure A.2: Target and the available services from the web-services domain [Felli, 2008]*

## A.2 Web Services Domain

Figure A.2 shows the complex web-services example [Felli, 2008]. It consists of five available services and a target. Each of them has ten states each.

# Appendix B

# Sample **Japex** Configuration file

```
<testSuite name="KR Example" xmlns="http://www.sun.com/japex/testSuite">
   <param name="libraryDir" value="lib"/>
   <param name="japex.classPath" value="${libraryDir}"/>
   <param name="japex.classPath" value="dist/classes"/>
   <param name="japex.numberOfThreads" value="1" />

        <param name="japex.resultUnit" value="ms"/>
        <param name="japex.warmupTime" value="30"/>
        <param name="japex.reportsDirectory" value="reports/"/>
        <param name="japex.chartType" value="linechart"/>
        <param name="japex.plotDrivers" value="true"/>
        <param name="japex.plotGroupSize" value="10"/>

    <driver name="RegressionDriver">
          <param name="Description" value="Regression based approach"/>
          <param name="japex.driverClass"
          value="japexDriver.RegressionDriver"/>
    </driver>
    <driver name="RegressionOptimizedDriver">
          <param name="Description" value="REG1 algorithm"/>
          <param name="japex.driverClass"
          value="japexDriver.RegressionOptimizedDriver"/>
    </driver>
    <driver name="RegressionOptimizedDriver2">
          <param name="Description" value="REGII algorithm"/>
          <param name="japex.driverClass"
          value="japexDriver.RegressionOptimizedDriver2"/>
    </driver>
```

```
    <driver name="ProgressionDriver">
          <param name="Description" value="Progression based approach"/>
          <param name="japex.driverClass"
          value="japexDriver.ProgressionDriver"/>
    </driver>
    <testCase name="test1">
          <param name="inputfile"
          value="testbed/NumberOfBehaviours/KR/KR1.xml"/>
          <param name="japex.runTime" value="00:05:00"/>
    </testCase>
    <testCase name="test2">
                <param name="inputfile"
                value="testbed/NumberOfBehaviours/KR/KR2.xml"/>
                <param name="japex.runTime" value="00:10:00"/>
    </testCase>
    <testCase name="test3">
                <param name="inputfile"
                value="testbed/NumberOfBehaviours/KR/KR3.xml"/>
                <param name="japex.runTime" value="00:15:00"/>
    </testCase>
    <testCase name="test4">
                <param name="inputfile"
                value="testbed/NumberOfBehaviours/KR/KR4.xml"/>
                <param name="japex.runTime" value="00:20:00"/>
    </testCase>
    <testCase name="test5">
                <param name="inputfile"
                value="testbed/NumberOfBehaviours/KR/KR5.xml"/>
                <param name="japex.runTime" value="01:00:00"/>
    </testCase>
</testSuite>
```

*Listing B.1: XML configuration for defining a test suite*

# Appendix C

# Extra Results

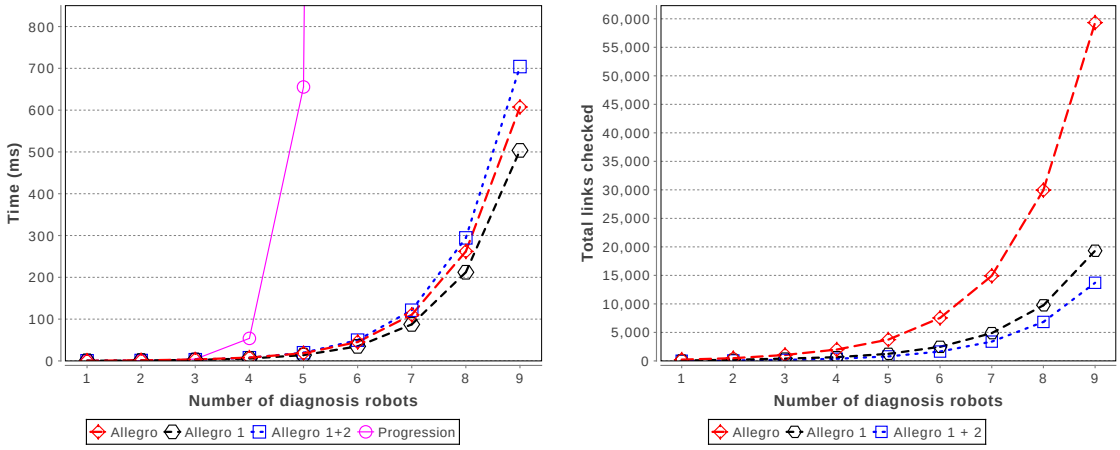


*Figure C.1: Results for Number of behaviors variation for Diagnostic robot of rescue domain*

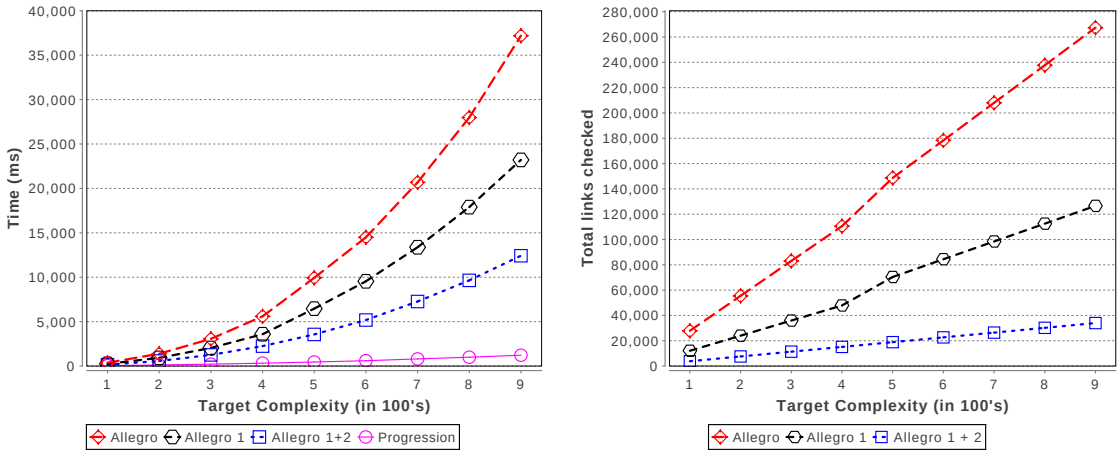


*Figure C.2: Results for Target complexity variation for rescue domain*

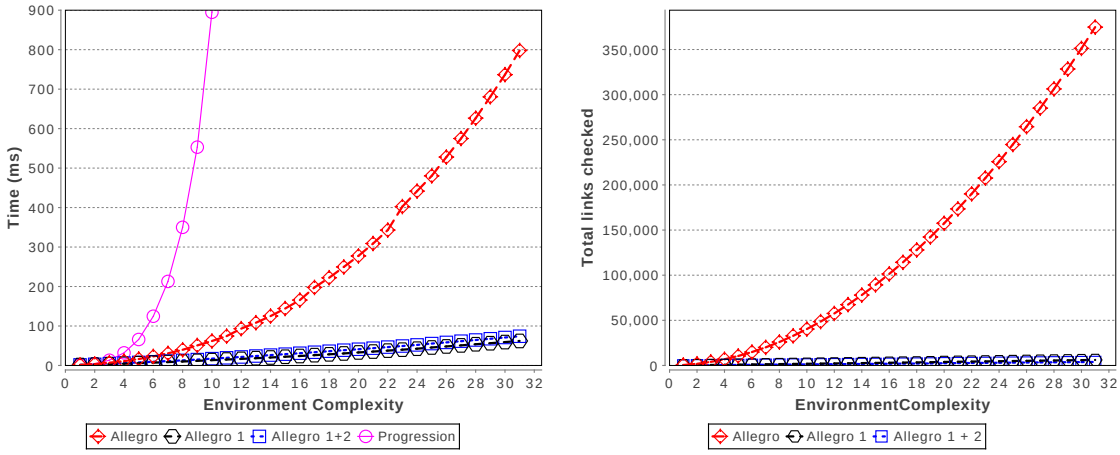


*Figure C.3: Results for Environment complexity variation for painting blocks domain*

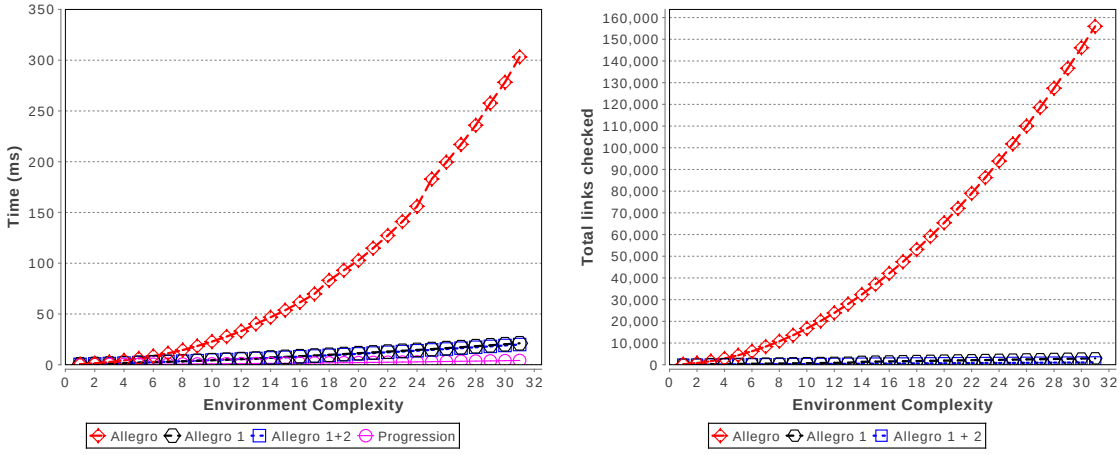


*Figure C.4: Results for Environment complexity variation for rescue domain*

# Bibliography

M. A. P. D. S. G. Almatelli Alessandro, Leoni Serafino. Symfony application for service integration, 2009. Universitá di Roma, Italy.

C. D. I. A. Q. H. Balestra Concetta, Coccia Chiara. Opus application for service integration, 2009. Universitá di Roma, Italy.

G. De Giacomo and S. Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In M. M. Veloso, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1866–1871, Hyderabad, India, Jan. 2007.

P. Felli. Service composition through synthesis techniques based on alternating-time temporal logic, 2008. Universitá di Roma, Italy.

M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Procedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 453–462, 1995.

M. Paleczny, C. Vick, and C. Click. The java hotspot TM server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, page 1. USENIX Association, 2001.

F. Patrizi. *Simulation-based Techniques for Automated Service Composition.* PhD thesis, Sapienza Universitá di Roma, Roma, Italy, 2008. Universitá di Roma, Italy.

S. Sardina and G. De Giacomo. Realizing multiple autonomous agents through scheduling of shared devices. In J. Rintanen and B. Nebel, editors, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 304–312, Sydney, Australia, Sept. 2008. AAAI Press.

S. Sardina, F. Patrizi, and G. De Giacomo. Automatic synthesis of a global behavior from multiple distributed behaviors. In R. C. Holte and A. Howe, editors, *Proceedings of the Na-*

*tional Conference on Artificial Intelligence (AAAI)*, pages 1063–1069, Vancouver, Canada, July 2007. Proceedings of the National Conference on Artificial Intelligence (AAAI).

S. Sardina, F. Patrizi, and G. De Giacomo. Behavior composition in the presence of failure. In G. Brewka and J. Lang, editors, *Proceedings of Principles of Knowledge Representation and Reasoning (KR)*, pages 640–650, Sydney, Australia, Sept. 2008. AAAI Press.

T. Stroeder and M. Pagnucco. Realising deterministic behaviour from multiple non-deterministic behaviours. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Pasadena, CA, USA, July 2009. AAAI Press.

L. Tan and R. Cleaveland. Simulation revisited. In *In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science (LNCS)*, pages 480–495, 2001.