Universiteit Leiden
Opleiding Informatica

Checking for Compatibility

in Team Automata

Name: Lau Bannenberg

Date: 28/08/2015

1st supervisor: Dr. H.C.M. Kleijn
2nd supervisor: Dr. H.J. Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Checking for Compatibility in Team Automata

BSc Computer Science Thesis

Lau Bannenberg

August 28, 2015

Supervisors **Dr. H. C. M. Kleijn** & **Dr. H. J. Hoogeboom**

Abstract

The Team Automata model allows us to reason about the communication and collaboration of systems consisting of both human and machine components. Recently [2] provided a rigorous definition for correct interaction among components. In this thesis we will provide means to test systems according to their compatibility requirements. We provide both a theoretical foundation and an implementation in Python. We look at the complexity and scalability of compatibility testing and examine several methods to manage the effort required, including the possibility of splitting the workload into distributed tasks.

Contents

1 Introduction **5**
- 1.1 Research Goals . . . 5
- 1.2 Organization of this paper . . . 6

2 Preliminaries **7**
- 2.1 Notation . . . 7
- 2.2 Definitions and theorems from earlier works . . . 8
  - 2.2.1 (Reactive) Transition Systems . . . 8
  - 2.2.2 Computations . . . 8
  - 2.2.3 Systems of RTSs . . . 8
  - 2.2.4 Synchronizations . . . 9
  - 2.2.5 Compatibility . . . 11
  - 2.2.6 Associativity of Synchronous Product . . . 12

3 Fixed Notation **15**

4 Main Contributions **17**
- 4.1 Determinism . . . 17
- 4.2 Finiteness . . . 17
- 4.3 Detecting livelock . . . 17
- 4.4 The straightforward approach: exhaustive search for directly incompatible states . . . 19
- 4.5 The backwards approach, starting in *DIS* . . . 21
- 4.6 Pruning the search space . . . 23
  - 4.6.1 Pruning unreachable states from the statespace . . . 25
  - 4.6.2 Pruning useless transitions . . . 25
  - 4.6.3 Pruning useless actions . . . 25
  - 4.6.4 Reducing components . . . 26
- 4.7 Constructing bottom-up from "correct" components . . . 26
  - 4.7.1 Using Lemma 30 . . . 30

5 Software **37**
- 5.1 Other software . . . 39

6 Experiments **41**

7 Conclusions **45**
- 7.1 Future Questions . . . 45

References **47**

Index **48**

A Code **50**

B Automata test sets **70**
- B.1 Boss/Employee . . . 70
- B.2 Coffee . . . 70
- B.3 Settlement . . . 71
- B.4 Token Ring . . . 72
- B.5 Wedding . . . 74
- B.6 Passing the Ball . . . 75

1 Introduction

We live in an era of systems that talk to each other. Subsystems designed by different people are used to assemble new systems, which will in turn be used in other systems. But with so many components interacting, can we predict if our systems will always be interacting correctly?

One interesting model for interacting components was introduced by Ellis in [4]. He introduced the idea of *team automata*. These resemble finite state automata, but with some key differences. The most significant of these is that team automata can perform some actions together, as a way of modeling communication (one component sending an instruction to another) and collaboration (components working together on an action as equals). A rigorous definition for the various ways components can act together was given by [7].

Another major property of team automata is that when we construct a team from these automata, the resulting team automaton can also be used as a component in further constructions. We can then construct a system as the product of teaming up subsystems that have subsystems of their own. This allows division of effort among developers and re-use of components.

Team automata can represent machine components of course, but also people interacting with machines, or even people interacting with each other without any machines at all. In our examples we will use a dysfunctional workplace, a coffee machine and customer, a groom who cannot make up his mind to marry, and a game of passing the ball around to show and test various aspects of team automata. We also have a few other cases that were useful for testing but are not ideal examples of any specific property. These are all included in the appendix.

Team automata can be used to reason about the correctness of the interaction of components. Carmona and Cortadella first defined the idea of Input/Output Compatibility for pairs of automata in [1]. Two main communication failures were to be avoided: message loss and deadlock. Message loss occurs when one automaton wants to send a signal and the other automaton is not ready to receive it. Deadlock occurs if both automata require input from one another before they can proceed themselves, leading too endless waiting. They gave a rigorous definition for when an assemblage of automatons could be called compatible. From this definition they suggested a decision procedure for determining compatibility, but did not elaborate on it.

In [2] Carmona and Kleijn lifted several of the restrictions of the model from [1], most significantly that compatibility was defined between a pair of automata. They instead produced a definition of compatibility for a system composed of an arbitrary number of automata. In addition, they went into more detail about the consequences of iterated team-building on compatibility. This was partially in an effort to arrive at an algorithm to determine whether a system was compatible. However, such an algorithm was not found.

That will be the objective of this thesis: to find an algorithm to determine the compatibility of a system of any number of automata, and analyze the complexity and scalability of the algorithm. Besides an abstract algorithm, we will also produce an implementation in Python.

1.1 Research Goals

The prime goal is to find an algorithm to decide whether a set of n components forms a compatible system.

- Analyze the algorithmic complexity of the solution.
- Try to keep the complexity manageable.
- Implement the algorithm and experiment with it. See if anything noteworthy happens.
- Analyze whether there is an advantage to analyzing a team through analyzing subteams and combining them, instead of analyzing the whole team at once.
- While we try several alterations to team automata to improve our algorithm, we must ensure the results remain valid. Specifically, we require the following properties to remain the same in any altered automaton:

    1. The *behaviour* exhibited by the alternate automaton must be the same as the original automaton. To an outside observer everything they can and can't do appears the same.

2. The alternate automaton is internally compatible if and only if the original automaton would have been compatible.
3. If the alternate automaton is used in an iterative process of team-building, it will have the same effects on its environment as the original. We will not attempt any optimization that will hinder us if we later decide to use the result as an ingredient for an even greater system.

- We will develop such an algorithm in theory, and also provide a Python implementation. In this implementation we will try to include such features as might be useful to a developer who wants to iteratively develop a system by designing and testing subsystems before integrating them.
- We aim to stay inside the limits already created by earlier works, notably [2], so that our results will be applicable in any case that meets their specifications.

1.2 Organization of this paper

We assume the reader is familiar with finite state automata and the basics of language theory, as well as sets. Because team automata are less well-known, we will introduce them and repeat some of the results from other authors to provide the necessary grounding for our own work.

We start in the **preliminaries** by introducing some notation and then revisit the definitions and theorems from (particularly) [2] that we build on. This section includes some examples of simple team automata to outline just what communication failures we wish to detect.

In our **main contributions** we develop a theoretical foundation for our algorithms, present our algorithms and analyze them. Then we study the possibilities for further improvement, first by looking at sharpening the theory.

In the **software** section we implement the algorithms in Python and explain their usage.

In the **experiments** section we do a few introductory experiments with heuristics to optimize our algorithms.

After **conclusions** follows an **appendix** with Python source code and specification of the components used in examples and experiments.

2 Preliminaries

2.1 Notation

Definitions and theorems will usually be introduced informally followed by a formal description. When a keyword is defined it will be ***bold and italicized***. Later references to it will be *italicized*.

We use the word 'component' to refer to a *reactive transition system* that is used as a building block in a *team automaton*. Since team automata can be used as the building blocks for new automata, components might have components of their own.

Definition 1 (*alphabet*)**.**

An **alphabet** Σ is a finite set of symbols. Σ^* denotes a set of strings of 0 or more symbols from Σ.

In this thesis we will commonly use Σ to denote an alphabet of *actions* of a *transition system*. Each symbol $a \in \Sigma$ represents one action of that TS.[1] We deviate from one common style of notation of languages: we will not use superscripts to indicate repetition, except with the Kleene star. For example, a^2 does not mean aa, but $(Q\Sigma)^*$ does indicate a concatenation of 0 or more strings consisting of an element from Q followed by an element of Σ.

When we take the Cartesian product of sets, we can recover elements from the ingredient sets with the **projection** function:

Definition 2 (*projection*)**.**

Let $V = V_1 \times V_2 \times \dots V_n$ be sets, $v = (v_1, v_2, \dots, v_n)$ and $\forall i \in \{1, \dots, n\} : v_i \in V_i$, then:

$\text{proj}_i(v) = v_i$

We will use subscripted indices to identify components (which are tuples) and the items that belong to them: $Q_{\mathcal{T}_i}$ is the set of states belonging to component $\mathcal{T}_i$. Usually we will use a shorthand, where we write Q_i instead of $Q_{\mathcal{T}_i}$. If the shorthand would prove confusing, we will use the full subscript instead. We will use superscripted incides to identify the individual items belonging to some set: q_i^1 and q_i^2 are states of Q_i. Although sets of states are not ordered (q^2 may be reachable before q^1), we will normally call the initial state q^0 (components start in an initial state).

When describing *transitions* from one state to another, we often describe it in the form $t = (q, a, q')$, where q is the current state, a is the action, and q' is the next state.

For subsets, we will use $A \subseteq B$ to indicate that A is a subset of B, possibly equal to B. If A is a proper subset of B we will write $A \subsetneq B$.

In pseudocode, we will use the following notations:

- $varname \leftarrow value$ assignment
- $[a^1, a^2, \dots, a^n]$ is a list can contain duplicate elements. We can query it for membership with $element \in listname$, and look at the first and last elements with $list.head$ and $list.tail$. We can add and remove elements from he head and tail: $listname.prepend(element)$, $listname.apppend(element)$, $listname.remo$ and $listname.removelast$; and an empty list can be represented as $[]$. When we access the i-th element of a list we write $listname_{[i]}$.
- $\{a^1, a^2, \dots, a^n\}$ is a set with unique elements. We can query it for membership with $element \in setname$ and add and remove elements with $setname.add(element)$ and $setname.remove(element)$. An empty set is represented as $\emptyset$
- Functions are written as $functionname(argument\ 1, argument\ 2, \dots, argument\ n)$.

For example, $func([a], b)$ is a call to function $func$ with as arguments a list containing just one element (a) and another element (b) that is not in a list.

[1] This alphabet could be empty, but that generates trivial automata that cannot change state nor in any way influence other automata. Therefore they have no impact on the compatibility of systems they are included in. It is worth noting that the material in this thesis applies to empty-alphabet automata just as much as it does to normal automata. This becomes relevant because some pruning techniques can remove actions from automata, possibly leaving their alphabets empty.

2.2 Definitions and theorems from earlier works

This thesis builds heavily on earlier works, and uses definitions, theorems and proofs presented there. They are repeated here, with attribution:

2.2.1 (Reactive) Transition Systems

From [2]

Reactive transition systems strongly resemble (finite) state automata. Their main difference is that in an RTS actions are classified on the individual automaton level as input, output or internal. One automaton might send "button!" as an output acton and another receive "button?" as input signal. Output actions represent a component taking action to influence its environment or to direct another component. Input actions represent the automaton reacting to such stimuli. Internal actions are hidden from the environment, happening entirely within the "black box".

There are two other noteworthy differences: the Team Automaton model does not prohibit infinite sets of states, although in practice this is rarely used. Also, there is no notion of a set of accepting states. An RTS often represents an ongoing process, unlike a finite state automaton that is trying to finish recognize a word.

Definition 3 (*transition system*)**.**

A **transition system** or **TS** is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I)$:
Q is a set of **states** , possibly infinite.
Σ is an *alphabet* of **actions** , with $Q \cap \Sigma = \emptyset$
$\delta \subseteq Q \times \Sigma \times Q$ is a set of (labeled) **transitions**.
$I \subseteq Q$ is a set of initial states.

A TS $\mathcal{A}$ is **reactive** (a **reactive transition system** or **RTS**) if its set of actions Σ is the union of three pairwise disjoint sets of input actions Σ^{inp}, output actions Σ^{out} and internal actions Σ^{int}. The union of input and output actions is the set of external actions Σ^{ext}.

If $\mathcal{A}$ is an RTS as above, then it can also be represented, with the explicit partition of its actions, as $(Q, (\Sigma^{inp}, \Sigma^{out}, \Sigma^{int}), \delta, I)$.

Graphically, output and input actions may be suffixed with the symbol ! and ? respectively to indicate their role, and internal actions can be indicated with the @ suffix.

For a TS $\mathcal{A} = (Q, \Sigma, \delta, I)$ and an action $a \in \Sigma$, the set of a-transitions in $\mathcal{A}$ is $\delta_a = \delta \cap (Q \times \{a\} \times Q)$.

Action a is said to be **enabled** in $\mathcal{A}$ at state $q \in Q$, denoted by a $\mathrm{EN}_{\mathcal{A}}$ q, if there exists a q' such that $(q, a, q') \in \delta$. We say that q is **input-enabled** if $\Sigma^{inp} \subseteq \{a | a \ \mathrm{EN}_{\mathcal{A}} \ q\}$.

2.2.2 Computations

From [2]

Definition 4 (*computation*)**.**

A **computation** of $\mathcal{A}$ is a (possibly infinite) sequence $\gamma = q^0 a^1 q^1 a^2 q^2 \ldots$ where $q^0 \in I$ and $q^i \in Q$ for all $1 \leq i$, and $(q^i, a^{i+1}, q^{i+1}) \in \delta$, for all $0 \leq i$.

A state q^i of $\mathcal{A}$ is **reachable** if it occurs in any computation of $\mathcal{A}$; in other words there is a series of transitions beginning in the initial state that eventually reaches q^i. All states that are not reachable are **unreachable**.

2.2.3 Systems of RTSs

From [2]

A *team automaton* is constructed from a finite number of components which are RTSs. These have to be composable with each other. For that to be true, their internal actions have to be truly internal;

the internal actions cannot also appear in the alphabet of other components of the system-to-be. This is because internal actions are private, not available for synchronization.[2]

Definition 5 (*composable system*)**.**

A collection $\mathcal{S} = \{\mathcal{A}_i | 1 \leq i \leq n\}$ of component automata (RTSs) $\mathcal{A}_i = (Q_i, \Sigma_i, \delta_i, I_i)$, with Σ_i partitioned into internal, input and output actions $(\Sigma_i^{int}, \Sigma_i^{inp}, \Sigma_i^{out})$, respectively, forms a **composable system** if $\Sigma_i^{inp} \cap \bigcup_{j=1, j\neq i} \Sigma_j = \emptyset$ for all $1 \leq j \leq n$.

Given a composable system, a TA is constructed by selecting *synchronizations* on common actions of its components, while retaining the local effect of each action. The state space Q of such a TA is $\Pi_{i=1}^n Q_i$. This means that each of a team automaton's states is a product of the states of its components. When it executes an action involving some of its components, those parts of the global state change. Components that do not participate in the action do not change state.

Definition 6 (*team automata*)**.**

Let $\mathcal{S}$ be a composable system as specified above. Let $a \in \bigcup_{i=1}^n \Sigma_i$.
The **complete transition space** of a in $\mathcal{S}$ is
$\Delta_a(\mathcal{S}) = \{(p, a, p') | \ p, p' \in \Pi_{i=1}^n Q_i, (\text{proj}_i(p), a, \text{proj}_i(p')) \in \delta_i$ for some $i \leq 1 \leq n$,
and for all $1 \leq i \leq n$ either $(\text{proj}_i(p), a, \text{proj}_i(p')) \in \delta_i$ or $\text{proj}_i(p) = \text{proj}_i(p')\}$.

A **team automaton** or **TA** over $\mathcal{S}$ is an
RTS $\mathcal{T} = (\Pi_{i=1}^n Q_i, \Sigma, \delta, \Pi_{i=1}^n I_i)$ with $\delta_a \subseteq \Delta_a(\mathcal{S})$ for all $a \in \Sigma^{ext}$ and $\delta_a = \Delta_a(\mathcal{S})$ for all $a \in \Sigma^{int}$.
Its set of actions is
$\Sigma = \bigcup_{i=1}^n \Sigma_i$ with $\Sigma^{int} = \bigcup_{i=1}^n \Sigma_i^{int}$, $\Sigma^{out} = \bigcup_{i=1}^n \Sigma_i^{out}$ and $\Sigma^{inp} = \bigcup_{i=1}^n \Sigma_i^{inp} - \bigcup_{i=1}^n \Sigma_i^{out}$.

Let $\mathcal{S} = \{A_i | 1 \leq i \leq n\}$ be a *composable system* as specified before. $\mathcal{S}$ represents a collection of interacting components. Let $a \in \bigcup_{i=1}^n \Sigma_i$ be an action of $\mathcal{S}$. Then dom($\mathcal{S}$, a) = $\{i | a \in \Sigma_i\}$ is the **domain** of a in $\mathcal{S}$. The **output domain** of a in $\mathcal{S}$ is outdom($\mathcal{S}$, a) = $\{i | a \in \Sigma_i^{out}\}$. Similarly, inpdom($\mathcal{S}$, a) = $\{i | a \in \Sigma_i^{inp}\}$ is the **input domain** of a in $\mathcal{S}$. We say that an external action a is **output-domain enabled** (**input-domain enabled**) at p if a $\text{EN}_{\mathcal{A}_i}$ $\text{proj}_i(p)$ for all $i \in$ outdom($\mathcal{S}$, a) (for all $i \in$ inpdom($\mathcal{S}$, a)). It is **domain enabled** if it is both input-domain and output-domain enabled.

An action is said to be **communicating** in $\mathcal{S}$ if it has a non-empty output domain and a non-empty input domain. We write $\Sigma^{com} = \bigcup_{i=1}^n \Sigma_i^{out} \cap \bigcup_{i=1}^n \Sigma_i^{inp}$. External actions which are not communicating in $\mathcal{S}$ may become so in an extension of $\mathcal{S}$ when components have been added. We say that $\mathcal{S}$ is **complete** if each of its external actions is communicating.

2.2.4 Synchronizations

From [7], [2]

As shown before, the transitions of a TA $\mathcal{T}$ over a composable system $\mathcal{S}$ form a subset of that system's complete transition space:

$$\delta_a \subseteq \Delta_a(\mathcal{S}) \text{ for all } a \in \Sigma^{ext} \text{ and } \delta_a = \Delta_a(\mathcal{S}) \text{ for all } a \in \Sigma^{int}$$

The latter means that δ inherits all the transitions based on internal actions of all its components. But let us focus on the fotmer now. It defines that δ_a can contain any of the possible combinations of transitions using a from the components of $\mathcal{T}$, but not necessarily all or even any of them.

For example, let us take $\mathcal{T}$ to be the TA over $\mathcal{S} = \{\mathcal{A}_1, \mathcal{A}_2\}$. Furthermore, $a \in \Sigma_1^{out}$ and $a \in \Sigma_2^{inp}$, and it is involved in the transitions $(q, a, q') \in \delta_{\mathcal{A}_1}$ and $(r, a, r') \in \delta_{\mathcal{A}_2}$. Now $\delta_{\mathcal{T}_a}$ could include a transition that moves both $\mathcal{A}_1$ and $\mathcal{A}_2$ to the next state. This would then be $((q, r), a, (q', r'))$. That would represent $\mathcal{A}_1$ sending a signal and $\mathcal{A}_2$ receiving it. But it could also include a transition that moves $\mathcal{A}_1$ but ignores $\mathcal{A}_2$, so $((q, x), a, (q', x))$. That would represent $\mathcal{A}_1$ sending a signal but $\mathcal{A}_2$ not responding to it.

[2] If two components happen to have internal actions with the same name, renaming those actions can make them composable. This could occur, for example, if a system is constructed with several identical components, such as a car with two front wheels. Although the wheels have the same sort of internal actions, those are still distinct actions, not shared – otherwise they would not be internal actions.

We call such a choice of component-transitions to use in a transition for the new TA a **synchronization**. It is possible to select synchronizations ad-hoc, picking some and not others, but normally we will use some rule that selects all synchronizations that meet some criterium, and excludes all others. Originally introduced in [4], this concept was rigorously developed in [7].

One obvious rule is that if an output-action happens in one or more components, all components able to process that action as input must also participate in the action. This would represent for example workers responding to the instructions of supervisors. Another, slightly less obvious rule is to require all components that possess a to participate in every execution of a by the team; if even one of them is not currently ready, a cannot execute. If they are all available, and a is executed, then thay all participate. This is sometimes called "action-indispensible" when the rule is applied to a single action, or "synchronous product" if it applies to all actions.

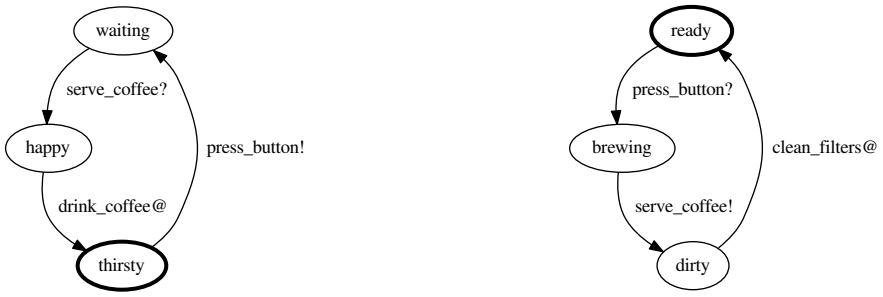
Definition 7 (*synchronous product*)**.**

Let the set of transitions $\chi^{\mathcal{S}}$ be defined by

$$\chi^{\mathcal{S}}_a = \{(p, a, p') \in \Delta_a(\mathcal{S}) \mid (\text{proj}_i(p), a, \text{proj}_i(p')) \in \delta_i \text{ for all } 1 \leq i \leq n \text{ such that } a \in \Sigma_i\} \text{ for all } a \in \Sigma$$

The **synchronous product** of a composable system $\mathcal{S}$, denoted by $\chi(\mathcal{S})$, is the TA over $\mathcal{S}$ with $\chi^{\mathcal{S}}$ as its set of transitions.

In [2] *compatibility* was defined specifically for synchronous product teams. In addition, it was shown there that many other synchronization schemes can be algorithmically converted to a synchronous product system while preserving their compatibility properties. Therefore in the rest of this work, we will only focus on determining compatibility for synchronous product automata.



(a) drinker (b) machine

Figure 1: A coffee drinker and a coffee machine.

In Figure 1 we see a coffee consumer and a machine. Their initial states are marked with the thicker line around the state. The consumer starts out thirsty, presses a button (for him an output action), and waits to receive coffee (input). He drinks it (an internal action) and then he becomes thirsty again. Meanwhile the coffee machine is waiting for orders when its button is pushed (input); it then serves coffee (output), and has to clean its filters (internal). When we take the *synchronous product* of these two components we get Figure 2, which shows us only the reachablee states of the system.

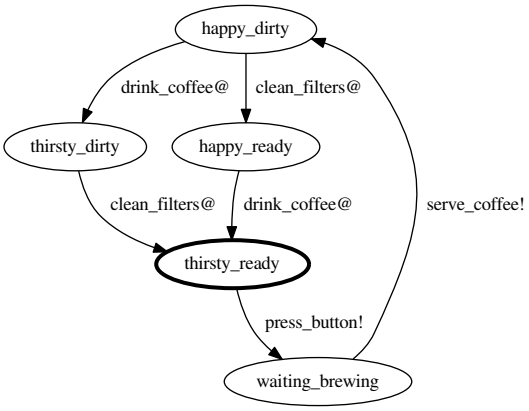


Figure 2: The synchronous product of the coffee drinker and the machine. Only reachable states and useful transitions are shown.

2.2.5 Compatibility

From [2]

An *RTS* $\mathcal{A}$ has a **livelock** if there is an infinite sequence $q^1, q^2, q^3, \ldots$ of states in Q and an infinite sequence $a^1, a^2, a^3, \ldots$ of internal actions in Σ such that $(q^i, a^i, q^{i+1}) \in \delta$ for all $i \geq 1$. A **livelock-free** RTS will always ultimately execute an external action or terminate.

As defined in [2], a composable system is compatible if there is a **compatibility relation** $\mathcal{R}$ between the states of the components of the new TA;

- The initial states are in the relation.
- Next-states reached by executing actions are in the relation.
- Receptiveness: for every communicating action, if all components that have it as an output action can currently execute ("send") it, then all components that have it as an input action must also be ready to execute ("receive") it.
- Deadlock-freeness: whenever the components of the input domain for a communicating action a are waiting for input, the system must be enabled for an action. (a does not need to be possible just yet, but *some* action must be.)

  (Also, the team will eventually take communicating actions, not just an endless series of internal actions. Therefore livelock-freeness is also required.)

It should be noted that this system can still end up in a situation where no more actions are possible, even though the above criteria hold. No internal action is enabled in any component, and for every external action the output domain and input domains are both not enabled. In this case, the system has reached a **terminating** state. A TA capable of reaching a terminating state is also called terminating. Termination is not considered a communications failure.

When applied to a system using *synchronous product*, this leads to the following definition:

Definition 8 (*compatibility*)**.**

Let $\mathcal{S}$ be a *composable system* as before. Then $\mathcal{R} \subseteq \Pi_{i=1}^{n} Q_i$ is a compatibility relation for $\mathcal{S}$ if $\Pi_{i=1}^{n} I_i \subseteq \mathcal{R}$ and for all $p \in \mathcal{R}$ the following conditions are satisfied.

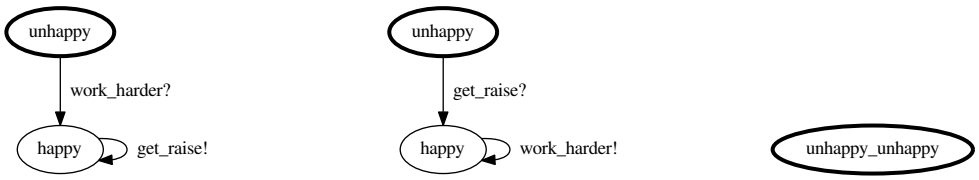
1. *Non-communicating Progress:* For all $a \in \bigcup_{i=1}^{n} \Sigma_i - \Sigma_{com}$, if a $\text{en}_{\mathcal{A}_i}$ $\text{proj}_i(p)$ for all $i \in dom(\mathcal{S}, a)$, then $p' \in \mathcal{R}$, whenever $(p, a, p') \in \chi_a^{\mathcal{S}}$.
2. *Receptiveness:* For all $a \in \Sigma_{com}$, if a is output-domain enabled at p, then a is input-domain enabled at p, and $p' \in \mathcal{R}$ whenever $(p, a, p') \in \chi_a^{\mathcal{S}}$.

3. *Deadlock-freeness:* If some action $a \in \Sigma_{com}$ is input-domain enabled at p, then there are $b \in \bigcup_{i=1}^{n} \Sigma_i$ and $p \in \Pi_{i=1}^{n} Q_i$ such that $(p, b, p') \in \chi^{\mathcal{S}}$.

$\mathcal{S}$ is said to be **compatible** if each of the component automata $\mathcal{A}_i$ is *livelock-free* and there exists a compatibility relation for $\mathcal{S}$. If a relation $\mathcal{R}$ fulfills conditions 1, 2, or 3, then it is called **progressive**, **receptive** or **deadlock-free**, respectively.

As examples of a *incompatible* systems, let us first consider the coffee machine and drinker from the previous section. Although the graph seems to suggest this system works fine, it does experience message loss. If the drinker finishes his coffee before the machine performs the 'clean_filters@' action, then he "can't" press the button. The definition of *synchronous product* requires that all components with the 'press button' action participate in it and the machine is not ready yet. What we have here is an example of message loss: it turns out this system fails the *receptiveness* requirement for compatibility.

As an example of deadlock, consider Figure 3 which shows an employee who refuses to work harder until he gets a raise, and a boss who refuses to give a raise until the employee works harder. Getting a raise is an input for the employee and output for the boss, and working harder is output for the employee and input for the boss.



(a) boss (b) employee (c) $\chi(\{boss, employee\})$

Figure 3: A deadlocked workplace

We should point out that in both the coffee and the workplace example, only reachable states were shown. If the employee somehow started out happy, then the system would not be deadlocked. And because synchronous product forbids actions unless all action-owners participate, we do not get to see what happens when the drinker pushes a button when the machine is not ready.

As an example of livelock, consider Figure 4 which shows us a groom-component (used in the "wedding" case found in the appendix). The groom has the possibility of endlessly hesitating before saying "I do." Such eternal doubting is clearly a communication failure at such an occasion.

2.2.6 Associativity of Synchronous Product

(The following results from [2] will not be used until section 4.7, but to clearly separate our own work from that of others, we introduce them here.)

We say that two RTSs $\mathcal{A}_1, \mathcal{A}_2$ are **action-complementary** if they have no common output/input actions, e.g., $\Sigma_1^{inp} \cap \Sigma_2^{inp} = \emptyset$ and $\Sigma_1^{out} \cap \Sigma_2^{out} = \emptyset$.

Furthermore, $\mathcal{A}_2$ **collaborates** with $\mathcal{A}_1$ if for every state of the Cartesian product, $(p_1, p_2) \in Q_1 \times Q_2$ and every $a \in (\Sigma_1^{inp} \cap \Sigma_2^{inp}) \cup (\Sigma_1^{out} \cap \Sigma_2^{out})$, a is enabled at p_2 whenever a is enabled at p_1.

Theorem 9. [2]
If $\mathcal{S} = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\}$ is *compatible* then $\mathcal{S}' = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ is *compatible*.

We repeat the proof of the following lemma here as a prelude to our discussion of a possible variant on it in 4.7.

Lemma 10. [2]
If $\mathcal{S} = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ is compatible and $\{\mathcal{A}_1, \mathcal{A}_2\}$ is *receptive* and *deadlock-free*, then $\mathcal{S}' = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\}$ is *compatible* provided that $\mathcal{A}_3$ *collaborates* with $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$.

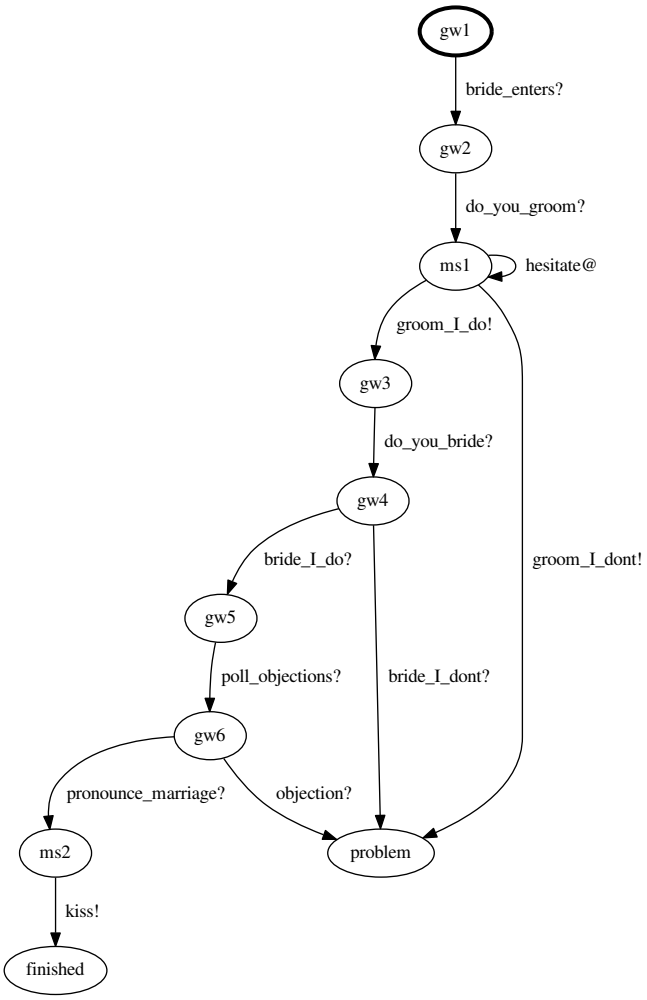


Figure 4: A groom who can't make up his mind.

Proof. Since internal actions do not communicate, $\chi(\mathcal{A}_1, \mathcal{A}_2\})$ is livelock-free if and only if $\mathcal{A}_1$ and $\mathcal{A}_2$ are both livelock-free.

Let $\mathcal{R}$ be a compatibility relation for $\mathcal{S}$. We will prove that $\mathcal{R}' = \{(p_1, p_2, p_3) | ((p_1, p_2), p_3) \in \mathcal{R}$ and (p_1, p_2) reachable in $\chi(\{\mathcal{A}_1, \mathcal{A}_2)\}$ is a compatibility relation for $\mathcal{S}'$.

Clearly, $\mathcal{R}'$ includes $I_1 \times I_2 \times I_3$.

Let $p = (p_1, p_2, p_3) \in \mathcal{R}'$ with $((p_1, p_2), p_3) \in \mathcal{R}$ and (p_1, p_2) reachable in $\chi(\{\mathcal{A}_1, \mathcal{A}_2)\}$.

Non-communicating progress: Let a be a non-communicating action of $\mathcal{S}'$ and assume that a is enabled at p_i whenever a is an action of $\mathcal{A}_i$ for $i = 1, 2, 3$. Let $(p, a, p') \in \chi_a^{\mathcal{S}}$ for some $p' = (p_1', p_2', p_3')$.

We have to prove that $p' \in \mathcal{R}'$ and (p_1', p_2') reachable in $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$.

Observe first that a is also a non-communicating action of $\mathcal{S}$. Moreover, in case a is an input (output) action of $\mathcal{A}_1$ or $\mathcal{A}_2$ or both, it is enabled at (p_1, p_2) in $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$. Consequently, a is enabled at $((p_1, p_2), p_3)$ and $(((p_1, p_2), p_3), a, ((p_1', p_2'), p_3')) \in \chi_a^{\mathcal{S}}$ independent of whether it is an input or output or internal action of $\mathcal{S}'$. Then by the non-communicating progress property of $\mathcal{R}$ it follows that $((p_1', p_2'), p_3') \in \mathcal{R}$ and hence $p' \in \mathcal{R}'$.

Receptiveness: Let a be a communicating action of $\mathcal{S}'$ and assume that a is enabled at p_i whenever a is an output action of $\mathcal{A}_i$ for $i = 1, 2, 3$.

We have to prove that a is also enabled for those $\mathcal{A}_i$ for which it is an input action and, moreover, if $(p, a, p') \in \chi_a^{\mathcal{S}'}$ for some $p' = (p_1', p_2', p_3')$, then $p' \in \mathcal{R}'$.

1. If a is not a communicating action of $\mathcal{S}$, then it must be the case that a is a communicating action of $\{\mathcal{A}_1, \mathcal{A}_2\}$ and a is not an input action of $\mathcal{A}_3$. The receptiveness of $\{\mathcal{A}_1, \mathcal{A}_2\}$ implies that

a is enabled at (p_1, p_2). Thus, a is enabled at the $\mathcal{A}_i$ for which it is an input action. Further, $(((p_1, p_2), p_3), a, ((p'_1, p'_2), p'_3)) \in \chi_a^{\mathcal{S}}$ and the non-communication progress property for $\mathcal{R}$ yields $((p'_1, p'_2), p'_3) \in \mathcal{R}$. Hence $p' \in \mathcal{R}'$.

2. If a is a communicating action of $\mathcal{S}$, then either a is a communicating action of $\{\mathcal{A}_1, \mathcal{A}_2\}$ and an input action of $\mathcal{A}_3$; or a is an output (input) action of either $\mathcal{A}_1$ or $\mathcal{A}_2$ or both, and an input (output) action of $\mathcal{A}_3$. In the first case we use again the receptiveness of $\{\mathcal{A}_1, \mathcal{A}_2\}$ together with the receptiveness of $\mathcal{R}$, to conclude that a is enabled at those $\mathcal{A}_i$ for which it is an input action and $(((p_1, p_2), p_3), a, ((p'_1, p'_2), p'_3)) \in \chi_a^{\mathcal{S}}$ with $((p'_1, p'_2), p'_3) \in \mathcal{R}$. Hence in this case $p' \in \mathcal{R}'$. For the latter case, the receptiveness of $\mathcal{R}$ suffices to finish the proof.

Deadlock-freeness: Let a be a communicating action of $\mathcal{S}'$ and assume that a is enabled at p_i whenever a is an input action of $\mathcal{A}_i$ for $i = 1, 2, 3$.

We have to prove that there exists an action b in $\mathcal{S}$ such that $(p, b, p') \in \chi^{\mathcal{S}'}$ for some $p' = (p'_1, p'_2, p'_3)$.

1. If a is an input action of $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ or an input action of $\mathcal{A}_3$, then the deadlock-freeness of $\mathcal{S}$ implies that there exist b and $((p'_1, p'_2), p'_3)$ such that $(((p_1, p_2), p_3), b, ((p'_1, p'_2), p'_3)) \in \chi^{\mathcal{S}}$ and we are done.

2. The remaining case is that a is communicating in $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ and not an input action of $\mathcal{A}_3$. Now we use the deadlock-freeness of $\{\mathcal{A}_1, \mathcal{A}_2\}$ to infer that there exists an action c and (p'_1, p'_2) such that $((p_1, p_2), c, (p'_1, p'_2)) \in \chi^{\{\mathcal{A}_1, \mathcal{A}_2\}}$. Again we have different cases to distinguish:

   If c is not an action of $\mathcal{A}_3$, we are done: $(p, c, (p'_1, p'_2, p'_3)) \in \chi^{\mathcal{S}'}$.

   The remaining cases are that (i) either c is an input action of $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ and an output action of $\mathcal{A}_3$ or (ii) c is an output action of $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ and an input action of $\mathcal{A}_3$ or (iii) c is an output (input) action of $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ and an output (input, respectively) action of $\mathcal{A}_3$.

   (i) As before, the deadlock-freeness of $\mathcal{S}$ implies that there exist $((p'_1, p'_2), p'_3)$ such that $(((p_1, p_2), p_3), c, ((p'_1, p'_2), p'_3)) \in \chi^{\mathcal{S}}$ and we are done.

   (ii) As before, the receptiveness of $\mathcal{R}$ implies that there exist b and $((p'_1, p'_2), p'_3)$ such that $(((p_1, p_2), p_3), b, ((p'_1, p'_2), p'_3)) \in \chi^{\mathcal{S}}$ and we are done.

   (iii) Due to the fact that $\mathcal{A}_3$ collaborates with $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$, there exists $((p'_1, p'_2), p'_3)$ such that $(((p_1, p_2), p_3), b, ((p'_1, p'_2), p'_3)) \in \chi^{\mathcal{S}}$ and we are done.

□

Lemma 11. [2]
The following implications hold:

- If $\{\mathcal{A}_1, \mathcal{A}_2\}$ is *complete*, then $\mathcal{A}_1, \mathcal{A}_2$ are *action-complementary*.
- If $\mathcal{A}_1, \mathcal{A}_2$ are *action-complementary*, then $\mathcal{A}_1$ *collaborates* with $\mathcal{A}_2$.

Theorem 12. [2]
If $\mathcal{S} = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ is *compatible* and $\{\mathcal{A}_1, \mathcal{A}_2\}$ is *receptive* and *deadlock-free*, then $\mathcal{S}' = \{\mathcal{A}_1, \mathcal{A}_2\mathcal{A}_3\}$ is *compatible* if either:

- $\mathcal{S}$ is *complete*
- $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ and $\mathcal{A}_3$ are *action-complementary*
- $\mathcal{A}_3$ *collaborates* with $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$

Corollary 13. [2]
If $\mathcal{S} = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ is a *compatible* and *complete* system and $\{\mathcal{A}_1, \mathcal{A}_2\}$ is *compatible*, then $\mathcal{S}' = \{\mathcal{A}_1, \mathcal{A}_2\mathcal{A}_3\}$ is *compatible*.

3 Fixed Notation

We now fix the following notation for the remainder of this thesis:

- $\mathcal{A} = (Q, \Sigma, \delta, I)$ is a *reactive transition system*.
- $\mathcal{S}$ is the *composable system* over $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$.
- $\mathcal{X}(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$ is the synchronous product of the *composable system* $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$
- $\mathcal{T}^i = \mathcal{X}(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i)$ where $1 \leq i \leq n$ and $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i \in \mathcal{S}$.
- $\mathcal{T} = \chi(\mathcal{S}) = \chi(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$.

Informally, $\mathcal{A}$ and $\mathcal{T}$ are both RTSs, however, $\mathcal{T}$ will only be used to refer to an RTS created as the *synchronous product* of other RTSs. If we want to describe a property that holds for any RTS, we will refer to $\mathcal{A}$. Likewise, we will use $\mathcal{A}_i$ to refer to the i-th component of $\mathcal{T}$.

$\mathcal{T}^i$ represents a team over the first i components of $\mathcal{S}$. As $\mathcal{S}$ has n components, $\mathcal{T} = \mathcal{T}^n$.

4 Main Contributions

4.1 Determinism

A deterministic TS has at most one initial state and for every state, every enabled action has exactly one next state.

Definition 14 (*determinism*)**.**

A *TS* is **deterministic** if the following conditions are both true:

1. $|I| \leq 1$
2. $\forall p \in Q, \forall a \in \Sigma : (p, a, q) \in \delta \wedge (p, a, q') \in \delta \implies q = q'$

If a TS is not deterministic it is **nondeterministic**.

For a deterministic TS we can represent the transitions of δ as a function:

$$\delta(q, a) = \begin{cases} q' & \text{if } (q, a, q') \in \delta \\ q & \text{otherwise} \end{cases}$$

The results of this thesis are applicable to nondeterministic automata as well, with the following adaptations. The transition function $\delta(current, action)$ would map to a set of next states rather than a single one. Everywhere in our algorithms where we evaluate a current state and action to determine the next state, we instead need to determine all possible next states and handle them. Lastly, we need to run our algorithms for all possible initial states.

So, while there is no theoretical barrier to using nondeterministic automata, for the purpose of clarity and ease of implementation in programming, we will restrict ourselves to deterministic automata.

4.2 Finiteness

The team automata concept allows automata with infinite states, but those pose additional problems for determining compatibility. Because compatibility is defined as a relation of states which all have certain properties, we might have to evaluate infinitely many states to decide whether the relation in fact exists. We will focus on the case of team automata with finite states. The algorithms used in this thesis are not guaranteed to terminate in the case of team automata with infinite states. In fact, they terminate only when only a finite number of states is actually reachable.

4.3 Detecting livelock

Definition 8 requires that components be livelock-free.

Lemma 15 (*persistence of livelock-freeness*)**.**

Each of $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ is livelock-free $\iff$ $\mathcal{T}$ is livelock-free.

Proof. Suppose $\mathcal{T}$ has a livelock. This means that starting in some state of $\mathcal{T}$ we can embark on an infinite series of transitions using only internal actions. However, the internal actions of the components of $\mathcal{T}$ do not affect each other's state. One component cannot enable a livelock in another component that did not exist in that component in isolation. So at least one of those components must have a livelock as well.

Now suppose $\mathcal{A}_i$ has a livelock. Since the state of the other components of $\mathcal{T}$ does not matter to the internal transitions $\mathcal{T}$ inherits from $\mathcal{A}_i$, this livelock continues to exist in $\mathcal{T}$.

□

Lemma 15 makes it possible to prevent livelock during a correct-by-construction bottom-up approach. We can verify for the initial components that they are livelock-free, and thus be certain that they will never cause a livelock when used later-on. We have chosen to insist on livelock-free components in all teams, so that we only need to test for livelocks on entirely new components, not components

created by sychronization. This is algorithmically cheaper (see below) and matches our iterative correct-by-construction philosophy.

It should be noted that we use the definition of livelock from [2], not [1]. A significant difference is that [1] requires livelock-freeness only in reachable parts of the transition space, while [2] requires it for the entire transition state. As this thesis is based strongly on [2] we will follow that definition and requirement. However, later on we will discuss pruning away unreachable states and transitions (see section 4.6), and when applied to an RTS with no unreachable states there is no practical difference between those definitions.

Algorithm 1 TryLivelock(list, action)

```
1:  current_state ← δ(list.tail, action)
2:  if current_state ∈ list then
3:     return true
4:  end if
5:  list.append(current_state)
6:  for all a ∈ Σ^int such that ∃next_state : (current_state, a, next_state) ∈ δ ∧ next_state ∉ safe do
7:     if TryLivelock(list, a) then
8:        return true
9:     end if
10: end for
11: safe.add(current_state)
12: list.removelast
13: return false
```

Algorithm 2 Detect Livelock

```
1:  global safe ← ∅
2:  for all q ∈ Q − safe do
3:     for all a ∈ Σ^int with ∃q' : (q, a, q') ∈ δ do
4:        if TryLivelock([q], a) then
5:           return true
6:        else
7:           safe.add(q)
8:        end if
9:     end for
10: end for
11: return false
```

Algorithm 2 calls Algorithm 1 on each state of the RTS $\mathcal{A}$ that has outgoing internal actions, provided that state has not been checked yet. Algorithm 1 then performs a depth-first search of the transition space starting in that state of $\mathcal{A}$ using only internal actions. Because (we required that) $\mathcal{A}$ has finite states, either the algorithm finds a repeat state, or terminates after reaching every other state but not finding a cycle. When TryLivelock exhausts its search space without finding a livelock, it returns False to DetectLivelock; and when DetectLivelock has exhausted its search space without finding a livelock it returns False to the caller. If a livelock is found by TryLivelock then it passes a True result upstream which eventually informs the caller that a livelock does exist in this component.

Since Algorithm 2 and Algorithm 1 maintain a global set called "$safe$" of states already found to be uninvolved in any livelocks, no state needs to be examined more than once. Unless a livelock is detected, every state will in fact be visited once. Again in the worst case, in every state except one, all internal actions are enabled, and lead towards other states in such a way that the internal transition space remains acyclic. In such a case, the complexity of this algorithm will be $|Q| \times |\Sigma^{int}|$ as it checks for enabled transitions in every state (TryLivelock line 6 and DetectLivelock line 3).

Since the size of the state space of a team of components $|Q| = |Q_1| \times |Q_2| \times \ldots \times |Q_n|$ is roughly exponential, and internal action alphabets do not overlap, the complexity of checking a team together would be:

$$\Big(|Q_1| \times |Q_2| \times \ldots \times |Q_n|\Big) \times \Big(|\Sigma_1^{int}| + |\Sigma_2^{int}| + \ldots + |\Sigma_n^{int}|\Big)$$

The complexity of checking the components separately is only:

$$\left(|Q_1| \times |\Sigma_1^{int}|\right) + \left(|Q_2| \times |\Sigma_2^{int}|\right) + \ldots + \left(|Q_n| \times |\Sigma_n^{int}|\right)$$

Clearly this is an argument in favor of preprocessing.

4.4 The straightforward approach: exhaustive search for directly incompatible states

To determine if $\mathcal{T}$ is *compatible*, we can use a straightforward algorithm to find a compatibility relation $\mathcal{R}$ for $\mathcal{T}$. If we can construct $\mathcal{R}$, then $\mathcal{T}$ is compatible. If we cannot construct $\mathcal{R}$, $\mathcal{T}$ must be compatible.

The definition of *compatibility* requires that there exists a compatibility relation; multiple relations are possible but at the very least the initial state must be in the relation, and any state directly reachable from a state in the relation must also be in the relation. Therefore, the smallest compatibility relation contains exactly the set of *reachable* states.

If we find a reachable state q that violates compatibility, through message loss or deadlock, then that state is not in the relation. We call this *directly incompatible* (see Definition 16). But since steps are not allowed to lead outside the relation, all states that can eventualy reach q must also not be in the relation. And since q was reachable from the initial state, the initial state is also not in the relation. Therefore, no compatibility relation exists for such a system.

Definition 16 (*directly incompatible state*)**.**

State $q \in Q$ is a **directly incompatible** state if one of the following holds:

1. $\exists a \in \Sigma^{com}$ is output-domain enabled in q, but not input-domain enabled in q.
2. $\exists a \in \Sigma^{com}$ is input-domain enabled in q, but there is no action that is enabled in q.

We will call the set of directly incompatible states **DIS**.

This then leads to the straightforward algorithm for determining the compatibility of $\mathcal{S}$: start in the initial state and find all states immediately reachable from there; put them in a queue. Visit each of them, finding more reachable states. In each of those states, test for direct incompatibility. If that is found, $\mathcal{S}$ is incompatible. If no direct incompatibility is found and all reachable states have been visited, the set of visited states $\mathcal{R}$ is now a minimal compatibility relation for $\mathcal{S}$.

Algorithm 3 will decide compatibility for finite, deterministic composable systems.[3] If it finds a contradiction it will terminate early[4], otherwise it has to run until all reachable states have been checked. However, is it a practical solution? Let us examine the complexity.

A reasonable measure of complexity is the number of times the enabledness of an action in a state is checked (lines 12–13 and 28). That is both one of the most frequent and costly computations of this algorithm. This is done for every action, in every reachable state of Q, unless the algorithm terminates early. Good cases include few reachable states and quickly-found contradictions. Bad cases include high amounts of reachable states and ironically, compatible systems that require the full exhaustive search to confirm.

Since $|Q| = |Q_1| \times |Q_2| \times \ldots \times |Q_n|$ and $|\Sigma| = |(\Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n)|$, we get a worst-case complexity of $O(|Q_i|^n)$, where $|Q_i|$ is average. Such exponential complexity may pose an obstacle to practical use of this algorithm. In the following sections, we will look at other approaches to try to handle this.

[3]The algorithm could easily be adjusted to also do so for nondeterministic systems as discussed before.

[4]This is not a technical necessity; the algorithm could instead just output a reference to the offending state, and continue searching, so that all offending states are found in one run of the algorithm.

Algorithm 3 Exhaustive State-Space Search

for all components $\mathcal{A}_1, \ldots \mathcal{A}_n$ **do**
  Use Algorithm 2 to verify that the component is livelock-free; if not, stop.
end for

$I \leftarrow [I_1 \ldots I_n]$
$Todo \leftarrow [I]$
$Done \leftarrow \emptyset$

while Todo $\neq$ [] **do**
  $current\_state \leftarrow Todo.head$
  $some\_progress \leftarrow$ **false**
  $awaiting\_input \leftarrow$ **false**

  for all $a \in \Sigma^{com}$ **do**
    if a is output domain enabled **then**
      if a is input domain enabled **then**
        $next\_state \leftarrow \delta(current\_state, a)$
        if $next\_state \notin Todo \wedge next\_state \notin Done$ **then**
          $Todo.append(next\_state)$
        end if
        $some\_progress \leftarrow$ **true**
      else
        CONTRADICTION: MESSAGE LOSS
      end if
    else
      if a is input domain enabled **then**
        $awaiting\_input \leftarrow$ **true**
      end if
    end if
  end for

  for all $a \notin \Sigma^{com}$ with $\wedge \forall i : a \in \Sigma_i \implies a \, \mathrm{EN}_{\mathcal{A}_i} current\_state_{[i]}$ **do**
    $next\_state \leftarrow \delta(current\_state, a)$
    if $next\_state \notin Todo \wedge next\_state \notin Done$ **then**
      $Todo.append(next\_state)$
    end if
    $some\_progress \leftarrow$ **true**
  end for

  if $some\_progress =$ **false** $\wedge$ $awaiting\_input =$ **true then**
    CONTRADICTION: DEADLOCK
  end if
  $Done.add(Todo.head)$
  $Todo.removefirst$
end while

4.5 The backwards approach, starting in *DIS*

Conceivably, only a small portion of the total state space of the system has compatibility problems. If this section is not reachable, then the system is fine. Since the previous approach experiences high complexity in systems that are compatible or close to it (having few states with direct incompatibility), it might be worthwhile to focus on the bad apples. If we can identify the states where a communication failure takes place, we can then try work backwards, to see if there is a path going to those states from the initial state.

We now define ultimately incompatible states as those that are directly incompatible, as well as those that can through a series of transitions arrive at a directly incompatible state.

Definition 17 (*ultimately incompatible state*)**.**

A state q is **ultimately incompatible** if one of the following holds:

1. $q \in DIS$
2. $\exists a, \exists q' : (q, a, q') \in \delta \wedge q' \in DIS$
3. $\exists a, \exists q' : (q, a, q') \in \delta \wedge q'$ is ultimately incompatible.

We will call the set of ultimately incompatible states **UDIS**.

Theorem 18 (*states in the compatibility relation are not in UDIS*)**.**

If $\mathcal{R}$ is a *compatibility relation*, then $q \in \mathcal{R} \implies q \notin UDIS$

Proof. Follows directly from Definition 8 which requires that the states of $\mathcal{R}$ do not suffer message loss or deadlock (i.e. not to be directly incompatible), and requires the same from all states reachable from it; versus Definition 17 which requires states in UDIS to have a path to a state that is directly incompatible. □

Lemma 19 (*I not in UDIS*)**.**

All components of $\mathcal{S}$ are livelock-free and $I \notin UDIS \implies \mathcal{S}$ is *compatible*.

Proof. Follows directly from Definitions 8 and 17. □

Lemma 19 suggests another straightforward method to decide the compatibility of $\mathcal{S}$;

1. Verify that all components of $\mathcal{S}$ are livelock-free.
2. Compute *DIS*.
3. Compute *UDIS*.
4. Verify that $I \notin UDIS$

Step 1 we accomplish with Algorithm 2, and step 4 is also simple. The intermediate steps are more complicated.

Let us consider the complexity of Algorithm 4. It performs two separate loops, computing states were receptiveness fails (1–6) and where there the system is deadlocked (7–15). To compute the states where receptiveness fails we must for every communicating output action a compute the states of the output domain where it is enabled, and then the states of the input domain where it is not enabled. To do so we need, for every component i, to go trough Q_i to see if a is enabled there, so we are now already looking at a complexity of $|\Sigma^{out} \cap \Sigma^{inp}| \times |Q_1| \times |Q_2| \times \ldots \times |Q_n|$.

Next we have the complexity of looking for deadlocks. Now we do this $|\Sigma^{inp}|$ times, and each such time first compute all enabled states of the input domain, and then for all possible states of the output domain, test *all* actions for enabledness. This gets us a complexity of $|\Sigma^{inp}| \times |Q_1| \times |Q_2| \times \ldots \times |Q_n| \times |\Sigma|$.

Algorithm 4 Compute DIS

Let $\mathcal{S}_{a!}$ be the output domain for a and let $\mathcal{S}_{a?}$ be the input domain for a.

$Q^{loss} \leftarrow \emptyset$
for all $a \in \Sigma^{out} \cap \Sigma^{com}$ **do**
  $Q_{a!} \leftarrow \{q | q \in Q_{\mathcal{S}_{a!}} \wedge a \text{ EN}_{\mathcal{S}_{a!}} q\}$ // Add q if the output domain is enabled there.
  $Q_{a?} \leftarrow \{q | q \in Q_{\mathcal{S}_{a?}} \wedge \neg(a \text{ EN}_{\mathcal{S}_{a?}} q)\}$ // Add q if the input domain is not enabled there.
  $Q^{loss}.add(Q_{a!} \times Q_{a?})$
end for

$Q^{lock} \leftarrow \emptyset$
for all $a \in \Sigma^{inp} \cap \Sigma^{com}$ **do**
  $Q_{a?} \leftarrow \{q | q \in Q_{\mathcal{S}_{a?}} \wedge a \text{ EN}_{\mathcal{S}_{a?}} q\}$ // Add q if the input domain is enabled. there.
  for all $p \in Q_{a?} \times Q_{\mathcal{S}_{a!}}$ **do**
    if $\neg(\exists b, \exists p' : (p, b, p') \in \delta)$ **then**
      $Q^{lock}.add(p)$ // Input expected, but no actions are enabled.
    end if
  end for
end for

$DIS \leftarrow Q^{loss} \cup Q^{lock}$

Algorithm 5 Compute UDIS

$UDIS \leftarrow \emptyset$
$todo \leftarrow [q | q \in DIS]$
while $todo \neq []$ **do**
  $current\_state \leftarrow todo.head$
  for all $\{p | p \in Q \wedge p \notin UDIS \wedge p \notin todo \wedge \exists a : (p, a, current\_state) \in \delta\}$ **do**
    $todo.append(p)$
  end for
  $UDIS.add(current\_state)$
  $todo.removefirst$
end while
return $UDIS$

The worst-case complexity of Algorithm 5 is disappointing. In the worst case we have to search, for every state found to be in UDIS, every state not yet in UDIS (or *todo*) to see if there is an action transitioning to the current state. Suppose we have a graph where the states essentially form a line from start to end, and the very last state is directly incompatible. Furthermore, every state is connected to the next one with every action (except for the final state; all we need to know about its outgoing transitions is that they result in direct incompatibility). Then for every state, beginning with the last one and working back to the beginning, we need to look up all the state/action combinations of all earlier states. This results in a complexity of:

$$\sum_{i=1}^{|Q|-1} (|Q| - i) \times |\Sigma|$$

This complexity is ostensibly not so bad, but keep in mind that $|Q| = |Q_1| \times |Q_2| \times \ldots |Q_n|$. So even without the added complexity of computing DIS, the worst-case complexity of the backwards approach is no better than that of the exhaustive forwards approach given earlier.

Given the difficulty of computing DIS and the wost-case complexity of computing UDIS, this approach is not feasible.

4.6 Pruning the search space

As the previous sections have shown, the size of the state space tends to explode when automata are teamed up. We will now investigate if it is possible to limit this explosion. We will examine what parts of a team automaton can be pruned without changing its actual functioning. To be admissible, a pruning method must meet these criteria:

1. The computations of the pruned automaton are the same as those of the original. (It has the same behaviour.)
2. The compatibility of the resulting automaton has to be the same as that of the original, regardless of what automata it is teamed up with.

   We do not optimize towards specific teamings with other components; we want the same results from a teaming with original or pruned automaton in every situation.

With these criteria, our pruned automaton remains a correct substitute in all future teams.

We have already discussed unreachable states of individual components. Likewise, a transition can be *useless* if it can never be used, and an action can be *useless* if there are no useful transitions using that action.

Definition 20 (*usefulness*)**.**

A transition $t \in \delta$ is **useful** if $t = (p, a, q)$ for some reachable state p. A transition that is not useful is **useless**.

An action a is useful if there is at least one useful $t \in \delta$ such that $t = (p, a, q)$. An action that is not useful is useless.

It would seem like we could prune away all these unreachable and useless parts of an automaton to make it smaller and faster to process. But we have to be sure that doing so does not change the way it functions, both in isolation, and in the context of a team.

Definition 21 (*behaviour*)**.**

The **behaviour** $\Gamma(\mathcal{A})$ of a *TS* $\mathcal{A}$ is the set of all *computations* of $\mathcal{A}$.

We now introduce *active computations*. These can be understood as the parts of a team's computations that a given component of that team actively plays a role in.

Definition 22 (*active computation*)**.**

If $\gamma = q^0 a^1 q^1 a^2 q^2 \cdots a^n q^n$ is a computation of $\mathcal{T}$, and $\mathcal{A}_i$ is the i-th component of $\mathcal{T}$, then γ_i^n is the **active computation** of $\mathcal{A}_i$ corresponding to γ, defined thus:

1. $\gamma_i^0 = \text{proj}_i(q^0)$
2. assume $\gamma_i^j = \omega p^j$ with $\omega \in (Q_i\Sigma_i)^*$, $p^j \in Q_i$, and $0 \leq j < n$
   then $\gamma_i^{j+1} = \begin{cases} \gamma_i^j a^{j+1} \text{proj}_i(q_{j+1}) & \text{if}(p^j, a^{j+1}, \text{proj}_i(q^{j+1})) \in \delta_i \\ \gamma_i^j & \text{otherwise} \end{cases}$

Definition 23 (*active behaviour*)**.**

The **active behaviour** $\Gamma_i^T(\mathcal{T})$ of a component $\mathcal{A}_i$ of $\mathcal{T}$ is the set of the active computations of $\mathcal{A}_i$ corresponding to the computations of $\mathcal{T}$.

So, how does the active behaviour of a component compare to its behaviour?

Theorem 24 (*active behaviour of a component in a team is a subset of its solitary behaviour*)**.**

$$\Gamma_i^T(\mathcal{T}) \subseteq \Gamma(\mathcal{A}_i)$$

Proof. All the active computations of $\mathcal{A}_i$ as defined by Definition 22 are also normal computations of $\mathcal{A}_i$ as defined by Definition 4. Therefore all active computations of $\mathcal{A}_i$ are also present in $\Gamma(\mathcal{A}_i)$. □

This does not immediately appear to be a very interesting result, but in the case of *synchronous product*, we can expand it a little, and then it does become very relevant to pruning;

Theorem 25 (*component's active behaviour in a team is a subset of active behaviour in subteams containing it*)**.**

Let $1 \leq i \leq j < k \leq n$. Then the following holds:

$$\Gamma_i^T(\mathcal{T}^k) \subseteq \Gamma_i^T(\mathcal{T}^j)$$

Proof. Suppose $\gamma_i \in \Gamma_i^{\mathcal{T}}(\mathcal{T}^i)$, but $\gamma_i \notin \Gamma_i^T(\mathcal{T}^j)$. Then γ_i must contain a sequence $(q, a, r) \in \delta_i$ where $\text{proj}_i(q)$ is not a *reachable* state of $\mathcal{A}_i$ in any computation of $\mathcal{T}^j$. However, q was a reachable state of A_i in isolation, because it occurs in γ_i. Thus q must have become unreachable as a result of applying Definition 7 (*synchronous product*). This necessarily implies that some other component $\mathcal{A}_x$ of $\mathcal{T}^j$ is not enabled for action a' whenever $\mathcal{A}_i$ is enabled for an action a' with $(p, a', q) \in \delta_i$. But if $\mathcal{A}_x \in \mathcal{T}^j$ then $\mathcal{A}_x \in \mathcal{T}^k$ and therefore $\gamma_i \notin \Gamma_i^T(\mathcal{T}^k)$ This proves that $\Gamma_i^T(\mathcal{T}^k) - \Gamma_i^T(\mathcal{T}^j) = \emptyset$.

The reverse does not necessarily hold however; it is possible that $\gamma_i \in \Gamma_i^T(\mathcal{A}^j)$ while $\gamma_i \notin \Gamma_i^T(\mathcal{A}^k)$. All that is required is that $\mathcal{A}_y$ with $j < y \leq k$ causes q to become unreachable in $\mathcal{A}_i$ as shown above. Therefore it is possible that $\Gamma_i^T(\mathcal{A}^j) \neq \Gamma_i^T(\mathcal{A}^k)$ □

Theorem 25 has some interesting implications for iterative construction of team automata. Once part of the behaviour of a component becomes inaccessible in a subteam, it will remain inaccessible in all teams that include that subteam.

Theorem 26 (*propagation of uselessness*)**.**

Let $1 \leq i \leq j < k \leq n$. Then the following holds:

1. Let $p \in Q_i$.
   If $\nexists q : q \in Q_{T^j}$ so that $\text{proj}_i(q) = p$ and q is reachable in $\mathcal{T}^j$,
   then $\nexists r : r \in Q_{T^k}$ so that $\text{proj}_i(r) = p$ and r is reachable.
2. Let $t = (p, a, p') \in \delta_i$.
   If $\nexists t' : t' = (q, a, q') \in \delta_{T^j}$ so that $(\text{proj}_i(q), a, \text{proj}_i(q')) = t$ and t' is useful,
   then $\nexists t'' : t'' = (r, a, r') \in \delta_{T^k}$ so that $(\text{proj}_i(k), a, \text{proj}_i(k')) = t$ and t' is useful.
3. If action a is useless in $\mathcal{T}^j$ then it is also useless in $\mathcal{T}^k$.

Proof.

1. If $\text{proj}_i(q)$ is unreachable in $\mathcal{A}^j$ then it is not part of any computation in $\Gamma_i^T(\mathcal{A}^j)$ By Theorem 25, it is therefore also not part of $\Gamma_i^T(\mathcal{A}^k)$, so it is also unreachable in $\mathcal{A}^k$.

2. If t is useless in $\mathcal{T}^j$ then by Definition 20 its current state is not reachable in $\mathcal{T}^j$. Then according to (1) its current state is also unreachable in $\mathcal{T}^k$, and so t is also useless in $\mathcal{T}^k$.

3. If a is useless in $\mathcal{T}^j$ then a exists in Σ_i of some component $\mathcal{A}_i$ of $\mathcal{T}^j$, but $\mathcal{A}_i$ has no useful transition using a as its action. According to (2) no useless transition of $\mathcal{A}_i$ can become useful in $\mathcal{T}^k$. Then under Definition 7 no synchronization on a can take place in $\mathcal{T}^k$, because $a \in \Sigma_i$, and therefore it must be part of any synchronization on a, but no useful transition from $\mathcal{A}_i$ can be found.

□

4.6.1 Pruning unreachable states from the statespace

As we have shown in Theorem 26, once a state becomes unreachable it will never become reachable again. We can therefore safely remove it from the component without changing either the behaviour of the component or the behaviour of any teams using that component. Note that Algorithm 3 implicitly does this, because its *Done*-set of states contains only reachable states.

4.6.2 Pruning useless transitions

Theorem 26 likewise informs us that it is safe to remove useless transitions. Since these transitions do not show up in behaviour, they can be pruned from a component without changing its behaviour.

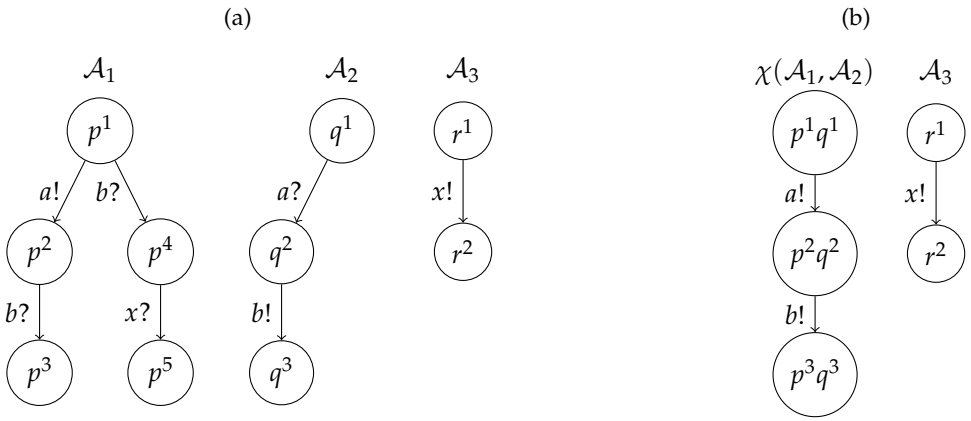
Because useless transitions are defined as originating in unreachable states, pruning unreachable states necessarily implies the pruning of useless transitions, because those transitions' originating state disappears; if $t = (q, a, q')$ and we prune q then we must also prune t.

4.6.3 Pruning useless actions

If an internal action is or becomes useless (because no transitions using it remain) then that action can be pruned as well, because its absence will not make any difference for the components behaviour or compatibility, in isolation or in a team.

It is *not* always safe to prune away useless external actions. Consider for example Figure 5; in $\mathcal{X}(\{\mathcal{A}_1, \mathcal{A}_2\})$ action $x?$ is useless. If we keep it, $\{\mathcal{X}(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ is not compatible, because receptiveness fails for $x!$. If we had discarded $x?$ then $\{\mathcal{X}(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ would have been compatible.

Figure 5: Useless actions still influence compatibility



The cause for this is the definition of *synchronous product*: if we want to execute a, all components with a in their alphabet need to participate in the execution. Because external actions that are currently noncommunicating might become so in the future, pruning them is not safe.

4.6.4 Reducing components

We can now formulate a way to reduce a component to its minimum size, without changing anything about its functioning, either in isolation or in a team. The result is a component which has only reachable states and only useful transitions. It can still have useless external actions, for reasons explained in section 4.6.3.

Definition 27. The **Reduced Reactive Transition System** (or **RRTS**) corresponding to the *RTS* $\mathcal{A}$ is another *RTS* $\mathcal{A}' = \langle Q', \Sigma', \delta', I' \rangle$:

- $Q' = \{q \in Q | q$
- $\Sigma' = \Sigma^{int\prime} \cup \Sigma^{out\prime} \cup \Sigma^{inp\prime}$
- $\Sigma^{int\prime} = \{a \in \Sigma^{int} | a$ is *useful*$\}$
- $\Sigma^{out\prime} = \Sigma^{out}$
- $\Sigma^{inp\prime} = \Sigma^{inp}$
- $\delta' = \{t \in \delta | t$ is *useful* $\}$
- $I' = I$

Clearly, there is only one RRTS corresponding to a given RTS. However, many RTSs might be reduced to the same RRTS. We can use Algorithm 6 to obtain an RRTS. The complexity of Algorithm 6 is based on the number of times the inner loop on lines 8–20 is executed. This is at worst $|Q| \times |\Sigma|$ because every action might be enabled in every state.

When we wish to explicitly denote that an RTS is an RRTS, we will indicate this with ρ. Likewise, we can indicate a variant of synchronous product that while creating a team using synchronous product, also reduces the resulting team, and we will write that as $\chi\rho()$.

Algorithm 6 ReduceRTS($\langle Q, (\Sigma^{int}, \Sigma^{out}, \Sigma^{inp}), I, \delta \rangle$)

```
1:  Todo ← [initial_state]
2:  Q' ← ∅
3:  Σ^out' ← Σ^out
4:  Σ^inp' ← Σ^inp
5:  δ' ← ∅
6:  while Todo ≠ [] do
7:      q ← Todo.head

8:      for all a ∈ Σ do
9:          if ∃q' ∈ Q : (q, a, q') ∈ δ then
10:             δ' ← (q, a, q')
11:             if q' ∉ Q' ∧ q' ∉ Todo then
12:                 Todo.append(q')
13:             end if
14:             if a ∈ Σ^int then
15:                 Σ^int' ← a
16:             end if
17:         end if
18:     end for

19:     Q'.add(Todo.head)
20:     Todo.removefirst
21: end while
22: return ⟨Q', (Σ^int', Σ^out', Σ^inp'), I', δ'⟩
```

4.7 Constructing bottom-up from "correct" components

The results of [2] referenced in section 2.2.6 demonstrate that (if certain conditions hold), it is possible to assemble a team in steps rather than all at once, while preserving its compatibility properties. Can we

use such an approach to cut down on the complexity of using Algorithm 3? If we have a set of automata that we want to form into a team, can we do so in steps rather than all at once, to reduce the workload?

For this to be even possible, we need to study the results of [2] and evaluate how practical their usage is. To begin with, Lemma 10 defines the relation between the compatibility of subteams and the final team only in a pairwise fashion; it provides no guidance on how to assemble a final team from three or more subteams, only from two subteams. This is because of the requirement that the second component *collaborates* with the first. This collaboration requirement is problematic for us, because it is not easy to test without actually building up the state space. So, to know if two automata collaborate and are therefore worth testing for compatibility (by computing their synchronous product) we have to compute their synchronous product.

The following lemma, theorem and corollary explore ways to circumvent this collaboration requirement. They use the fact that *action-complementary* components automatically collaborate, and that if a pair of automata form a complete system, they must be action-complementary. These results are not satisfactory however. It is true that all action-complementary pairs of components collaborate, because they have no shared actions that can fail the test. But relying on action-complementarity means we can no longer handle collaborating pairs that do share actions. To be able to use the much simpler test of action-complementarity we sacrifice coverage of the whole problem. Completeness is an even narrower set of automata, so we lose even more coverage.

But even the collaborating approach is not really good enough. We would also like to cover systems that cannot be subdivided into collaborating pairs. Can we do that? We need to take a look at where the requirement for collaboration in Lemma 10 comes from. The proof of the Lemma is quite long, but uses collaboration only near the end, to handle the last case.

We will now propose a variation on Lemma 10 that dispenses with the collaboration requirement. This produces a more generic rule that we can use to write a recursive algorithm for determining the compatibility of $\mathcal{S}$.

Lemma 28 (*correspondence of transitions*)**.**

Let $1 < k < n$
Let $\mathcal{T} = \chi(\{\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}), \mathcal{A}_{k+1}, \ldots, \mathcal{A}_n\})$
Let $\mathcal{T}' = \chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\})$
Then:

$$(((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n), a, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)) \in \delta_{\mathcal{T}} \iff ((p_1, \ldots, p_n), a, (p'_1, \ldots, p'_n)) \in \delta_{\mathcal{T}'}$$

Proof. Observe that if $t = (((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n), a, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n))$ is a transition of $\mathcal{T}$ then for all $i = 1, \ldots, n$, (p_i, a, p'_i) is a transition of $\mathcal{A}_i$ or a is not an action of $\mathcal{A}_i$.

Likewise, if $t' = ((p_1, \ldots, p_n), a, (p'_1, \ldots, p'_n)$ is a transition of $\mathcal{T}'$, then for $i = 1, \ldots, n$, (p_i, a, p'_i) is a transition of $\mathcal{A}_i$ or a is not an action of $\mathcal{A}_i$.

Recall that by Definition 7 every component $\mathcal{A}_i$ of a team made with synchronous product that has a in its alphabet participates in every transition using a or else a cannot be executed at that state.

If $\mathcal{T}$ can execute t, then all participating components are enabled for a in that state of $\mathcal{T}$. As a consequence, all those components are also able to participate in the state of $\mathcal{T}'$ that has the same local states for the components.

If $\mathcal{T}$ executes $(((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n), a, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n))$ and $\mathcal{A}_i$ has $a \in \Sigma_i$, $\mathcal{A}_i$ participates. If $\mathcal{T}'$ executes a as well, then $\mathcal{A}_i$ would participate as well. Likewise if $\mathcal{T}$ shadows $\mathcal{T}'$. □

Lemma 29 (*correspondence of states*)**.**

Let $1 < k < n$
Let $\mathcal{T} = \chi\rho(\{\chi\rho(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}), \mathcal{A}_{k+1}, \ldots, \mathcal{A}_n\})$
Let $\mathcal{T}' = \chi\rho(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\})$
Then $(p_1, \ldots, p_k), p_{k+1}, \ldots p_n) \in Q_{\mathcal{T}} \iff (p_1, \ldots, p_n) \in Q_{\mathcal{T}'}$

Proof. Follows directly from Lemma 28 and Definition 27 as follows:

Observe that $(I_1 \times \ldots \times I_k) \times I_{k+1} \ldots \times I_n) \in Q_{\mathcal{T}})$ and $(I_1 \times \ldots \times I_n) \in Q_{\mathcal{T}'}$

Assume $(p_1, \ldots, p_k), p_{k+1}, \ldots p_n) \in Q_{\mathcal{T}}$. Because $\mathcal{T}$ and $\mathcal{T}'$ are reduced, $(p_1, \ldots, p_k), p_{k+1}, \ldots p_n)$ is reachable. If we apply Lemma 28 we know that $(p_1, \ldots, p_n)$ is therefore also reachable in $\mathcal{T}'$

Then if $(((p_1,\ldots,p_k),p_{k+1},\ldots,p_n),a,((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n)) \in \delta_{\mathcal{T}}$, therefore $((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n) \in Q_{\mathcal{T}}$ and according to Lemma 28 $((p_1,\ldots,p_n),a,(p'_1,\ldots,p'_n)) \in \delta_{\mathcal{T}'}$, and therefore $(p'_1,\ldots,p'_n) \in Q_{\mathcal{T}'}$.

The reverse implication works in the same way. □

For a set of components $\{\mathcal{A}_2,\ldots,\mathcal{A}_n\}$, we say that it **collaborates on output** with $\mathcal{A}_1$ if for every state of the Cartesian product, $(p_1,p_2,\ldots p_n) \in Q_1 \times Q_2 \times \ldots \times Q_n$, and every $a \in (\Sigma_1^{out} \cap (\Sigma_2^{out} \cup \ldots \cup \Sigma_n^{out}))$, a is enabled at $(p_2,\ldots,p_n)$ whenever a is enabled at p_1.

Lemma 30 (*correspondence of compatibility*)**.**

Let $1 < k < n$.
Let $\mathcal{S} = \{\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}),\mathcal{A}_{k+1},\ldots,\mathcal{A}_n\}$ be compatible;
Let $\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}$ is *receptive* and *deadlock-free*;
If $\mathcal{A}_{k+1},\ldots,\mathcal{A}_n$ collaborates with every output action of $\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_k\})$,
then $\mathcal{S}' = \{\mathcal{A}_1,\ldots,\mathcal{A}_n\}$ is *compatible*.

Proof. Since $\mathcal{S}$ is compatible, by Definition 8 each of $\chi(\{\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}),\mathcal{A}_{k+1},\ldots,\mathcal{A}_n\})$ is livelock-free. And by Lemma 15 that means that $\mathcal{A}_1,\ldots,\mathcal{A}_n$ are all livelock-free. And again by Lemma 15 that means $\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_n\})$ is livelock-free.

Let $\mathcal{R}$ be a compatibility relation for $\mathcal{S}$. We will prove that

$$\mathcal{R}' = \{(p_1,\ldots,p_n) | ((p_1,\ldots,p_k),p_{k+1},\ldots,p_n) \in \mathcal{R} \text{ and } (p_1,\ldots p_k) \text{ reachable in } \chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_k\})\}$$

is a compatibility relation for $\mathcal{S}'$.

Clearly, $\mathcal{R}'$ includes $I_1 \times \ldots \times I_n$.

Let $p = (p_1,\ldots,p_n) \in \mathcal{R}'$ with $((p_1,\ldots,p_k),p_{k+1},\ldots,p_n) \in \mathcal{R}$ and $(p_1,\ldots p_k)$ reachable in $\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_k\})$.

Non-communicating progress: Let a be a non-communicating action of $\mathcal{S}'$ and assume that a is enabled at p_i whenever a is an action of $\mathcal{A}_i$ for $i = 1,\ldots,n$. Let $(p,a,p') \in \chi_a^{\mathcal{S}'}$ for some $p' = (p'_1,\ldots,p'_n)$.

We have to prove that $p' \in \mathcal{R}'$ and $(p'_1,\ldots,p'_k)$ reachable in $\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_k\})$.

Observe first that a is also a non-communicating action of $\mathcal{S}$. Moreover, in case a is an input (output) action of one or more of $\mathcal{A}_1,\ldots,\mathcal{A}_n$, it is enabled at $(p_1,\ldots,p_n)$ in $\chi(\{\mathcal{A}_1,\ldots,\mathcal{A}_n\})$. Consequently, a is enabled at $((p_1,\ldots,p_k),p_{k+1},\ldots,p_n)$ and $(((p_1,\ldots,p_k),p_{k+1},\ldots,p_n),a,((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n)) \in \chi_a^{\mathcal{S}}$ independent of whether it is an input or output or internal action of $\mathcal{S}'$. Then by the non-communicating progress property of $\mathcal{R}$ it follows that $((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n) \in \mathcal{R}$ and hence $p' \in \mathcal{R}'$.

Receptiveness: Let a be a communicating action of $\mathcal{S}'$ and assume that a is enabled at p_i whenever a is an output action of $\mathcal{A}_i$ for $i = 1,\ldots,n$.

We have to prove that a is also enabled for those $\mathcal{A}_i$ for which it is an input action and, moreover, if $(p,a,p') \in \chi_a^{\mathcal{S}'}$ for some $p' = (p'_1,\ldots,p'_n)$, then $p' \in \mathcal{R}'$.

If a is not a communicating action of $\mathcal{S}$, then it must be the case that a is a communicating action of $\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}$ and a is not an input action of any of $\mathcal{A}_{k+1},\ldots,\mathcal{A}_n$. The receptiveness of $\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}$ implies that a is enabled at $(p_1,\ldots,p_k)$. Thus, a is enabled at the $\mathcal{A}_i$ for which it is an input action. Further, $(((p_1,\ldots,p_k),p_{k+1},\ldots,p_n),a,((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n)) \in \chi_a^{\mathcal{S}}$ and the non-communication progress property for $\mathcal{R}$ yields $((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n) \in \mathcal{R}$. Hence $p' \in \mathcal{R}'$.

If a is a communicating action of $\mathcal{S}$, then we have cases:

1. a is a communicating action of $\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}$ and an input action of one or more of $\mathcal{A}_{k+1},\ldots,\mathcal{A}_n$; in this case we use again the receptiveness of $\{\mathcal{A}_1,\ldots,\mathcal{A}_k\}$ together with the receptiveness of $\mathcal{R}$ to conclude that a is enabled at those $\mathcal{A}_i$ for which it is an input action and $(((p_1,\ldots,p_k),p_{k+1},\ldots,p_n),a,((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n)) \in \chi_a^{\mathcal{S}}$ with $((p'_1,\ldots,p'_k),p'_{k+1},\ldots,p'_n) \in \mathcal{R}$. Hence in this case $p' \in \mathcal{R}'$.

2. a is an output action of one or more of $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ but not an input action of any of them, and an input (and possibly output) action of one or more of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$. In this case the receptiveness of $\mathcal{R}$ suffices to finish the proof.

3. a is an input (but not output) action of one or more of $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$, and an output (and possibly input) action of one or more of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$. In this case the receptiveness of $\mathcal{R}$ suffices to finish the proof.

4. a is not an action of any of $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$, but is an input action in one or more of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$ and also an output action in one or more of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$. Again the receptiveness of $\mathcal{R}$ suffices to finish the proof.

Deadlock-freeness: Let a be a communicating action of $\mathcal{S}'$ and assume that a is enabled at p_i whenever a is an input action of $\mathcal{A}_i$ for $i = 1, \ldots, n$.

We have to prove that there exists an action b in $\mathcal{S}$ such that $(p, b, p') \in \chi^{\mathcal{S}'}$ for some $p' = (p'_1, \ldots, p'_n)$.

1. If a is an input action of $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ or an input action of one or more of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$, then the deadlock-freeness of $\mathcal{S}$ implies that there exist b and $((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)$ such that $(((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n), b, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)) \in \chi^{\mathcal{S}}$ and we are done.

2. The remaining case is that a is communicating in $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ and not an input action of any of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$. Now we use the deadlock-freeness of $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ to infer that there exists an action c and $(p'_1, \ldots, p'_k)$ such that $((p_1, \ldots, p_k), c, (p'_1, \ldots, p'_k)) \in \chi^{\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}}$. Again we have different cases to distinguish:

   If c is not an action of any of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$, we are done: $(p, c, (p'_1, \ldots, p'_n)) \in \chi^{\mathcal{S}'}$.

   The remaining cases are that:

   (a) c is an input action of $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ and an output (and possibly input) action of some or all of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$.
   As before, the deadlock-freeness of $\mathcal{S}$ implies that there exist $((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)$ such that $(((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n)), c, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)) \in \chi^{\mathcal{S}}$ and we are done.

   (b) c is an output action of $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ and an input action of some or all of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$, but not an output action of any of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$.
   As before, the receptiveness of $\mathcal{R}$ implies that there exist b and $((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)$ such that $(((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n)), b, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)) \in \chi^{\mathcal{S}}$ and we are done.

   (c) c is an input action of $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ and an input action of some or all of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$, and not an output action of any of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$. Again, the deadlock-freeness of $\mathcal{S}$ implies that there exist $((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)$ such that
   $(((p_1, \ldots, p_k), p_{k+1}, \ldots, p_n)), c, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots, p'_n)) \in \chi^{\mathcal{S}}$ and we are done.

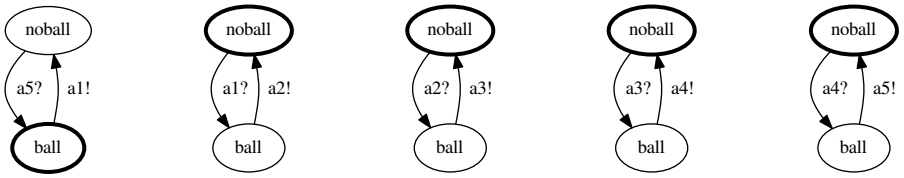
   (d) c is an output action of $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ and an output action of some or all of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$ and possibly an input action of any of $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$.
   Because of the fact that $\mathcal{A}_{k+1}, \ldots, \mathcal{A}_n$ collaborates with $\chi(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\})$ for output actions, there exists $((p'_1, \ldots, p_k), p'_{k+1}, \ldots p'_n)$ such that
   $(((p_1, \ldots, p_k), p_{k+1}, \ldots p_n), c, ((p'_1, \ldots, p'_k), p'_{k+1}, \ldots p'_n)) \in \chi^{\mathcal{S}}$ and we are done.

□

Compared to Lemma 10 this lemma shows a few improvements; it narrows the scope of collaboration that is required to only collaboration on output actions, and it generalizes it to systems with a "leader" of length ≥ 1, with ≥ 1 "followers". However, we were not able to drop the collaboration completely. Still, this new lemma also enables us to break down systems of $n > 3$ into smaller compatibility testing jobs instead of having to test everything at once.

4.7.1 Using Lemma 30

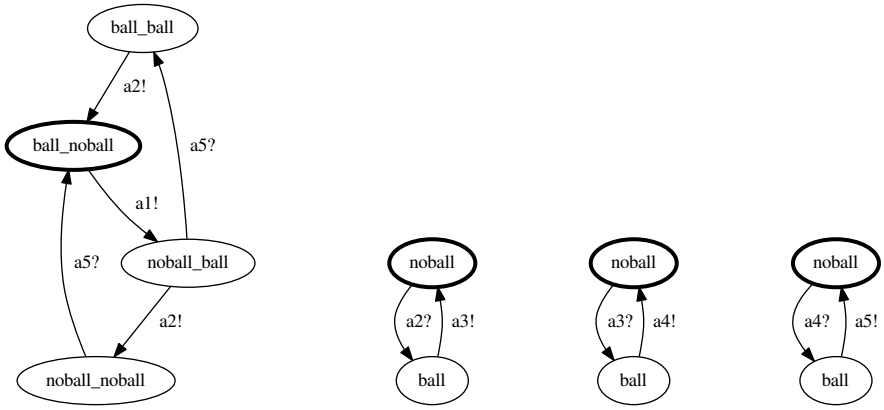
The implication of Lemma 30 is that we can take some elements out of $\mathcal{S}$, provided the rest collaborates on output with them; see if the chosen elements are compatible, and if so replace them in $\mathcal{S}$ with their synchronous product. For example, let's take a system $E = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$ which represents five people passing a ball around in a clockwise manner, shown in Figure 6 They each have an input action to catch a ball from their right neighbour, and an output action to throw it to their left neighbour.



(a) $\mathcal{A}_1$ (b) $\mathcal{A}_2$ (c) $\mathcal{A}_3$ (d) $\mathcal{A}_4$ (e) $\mathcal{A}_5$

Figure 6: Five players passing around a ball: $E = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$

It is immediately obvious that in this system, each of the components enjoys the output collaboration of the other components, because they do not share any output actions. We will now demonstrate how Lemma 30 allows us to decide the compatibility of our example system step by step. First we pick the first two players and test them for compatibility using Algorithm 3. Having found that they are compatible, we then replace $\mathcal{A}_1$ and $\mathcal{A}_2$ with $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ so that $E' = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$. Shown in Figure 7:



(a) $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ (b) $\mathcal{A}_3$ (c) $\mathcal{A}_4$ (d) $\mathcal{A}_5$

Figure 7: Five players passing around a ball: $E' = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$

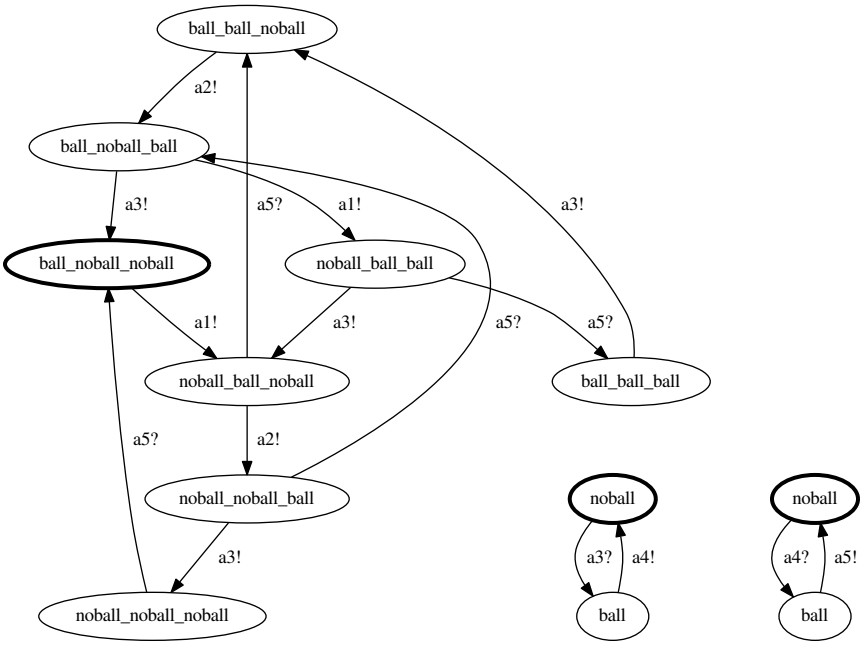
We don't know yet if E' is compatible, but if it is, then by Lemma 30 so is E. Note that in this intermediate step, there is a state where both components have a ball. So we take two components out of E' and repeat the test. From all the possible choices, we happen to prefer $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ and $\mathcal{A}_3$. They are compatible and so $E'' = \{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \mathcal{A}_5\}$. Shown in Figure 8:

This system now has a component that has many more states than the previous components. In fact it has $(|Q_1| \times |Q_2|) \times |Q_3| = 8$ states. And we are not done yet. We add $\mathcal{A}_4$, to obtain Figure 9:

Even bigger. But then we add in the fifth component (Figure 10):

Now we trace our way back to the original system:

- The final appplication of our algorithm tells us that $E''' = \{\chi(\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \}), \mathcal{A}_5\}$ is compatible.



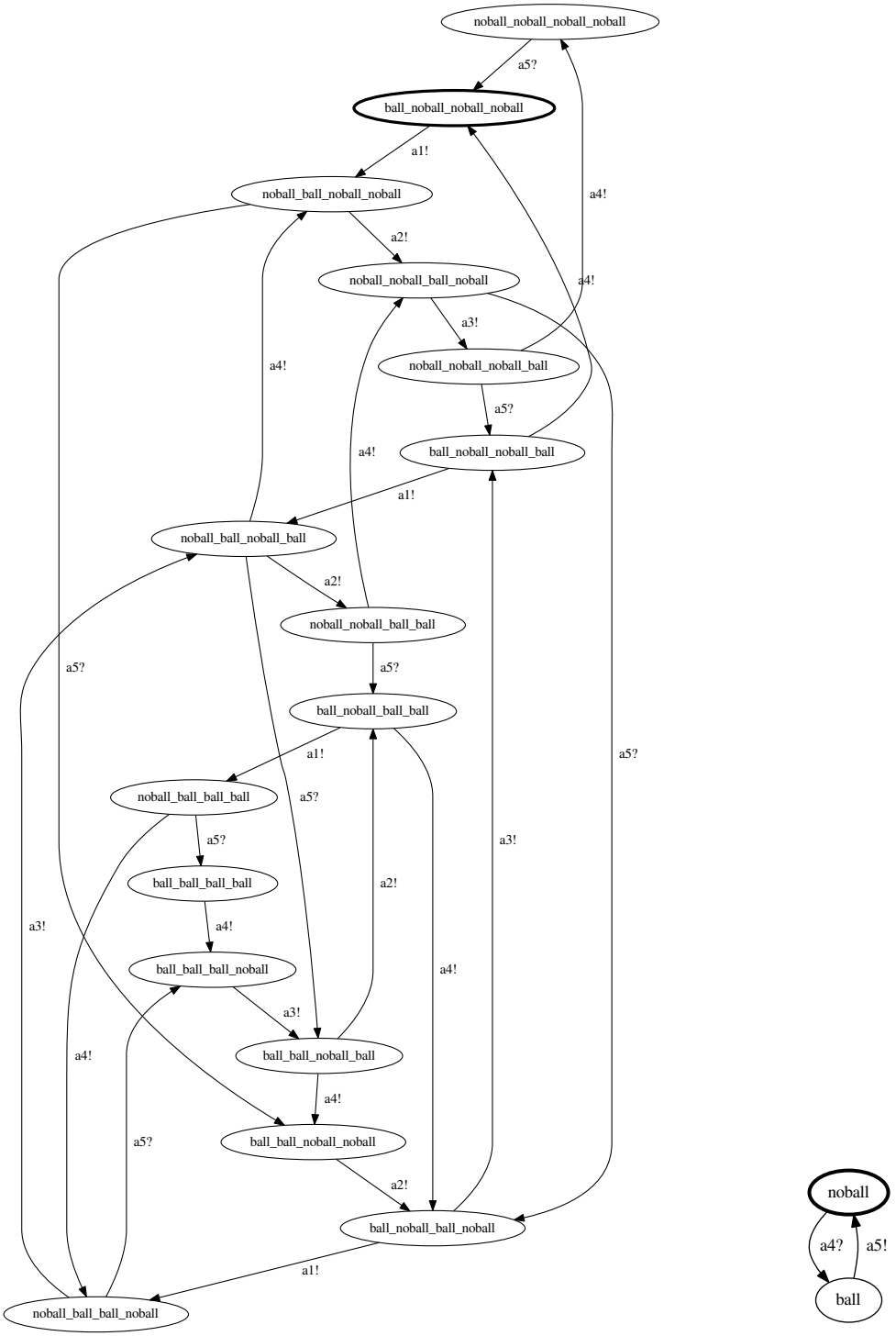
(a) $\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\})$ (b) $\mathcal{A}_4$ (c) $\mathcal{A}_5$

Figure 8: Five players passing around a ball: $E'' = \{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \mathcal{A}_5\}$

- Since we knew that $\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4\}$ is compatible and $\mathcal{A}_5$ collaborates with $\chi(\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4\})$ on output, by Lemma 30, that means that $E'' = \{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \mathcal{A}_5\}$ is compatible.
- Then because we know that $\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \mathcal{A}_5\}$ and $\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}$ are compatible and that $\mathcal{A}_4$ and $\mathcal{A}_5$ collaborate on output with $\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\})$, we know that $E' = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$ is compatible.
- And then because we know that $\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$ and $\{\mathcal{A}_1, \mathcal{A}_2\}$ are compatible and that $\mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5$ collaborate on output with $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$, we can conclude that $E = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$ is compatible.

Lemma 30 doesn't force us to assemble the system in this particular order; we could have also tried to combine the $\mathcal{A}_3$ with $\mathcal{A}_4$ and then combined $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ with $\chi(\{\mathcal{A}_3, \mathcal{A}_4\})$, as shown in Figure 11 and Figure 12.

Although Figure 9 and Figure 12 look different, they have the same amount of states and the remaining synchronization in Figure 12 results in a TA identical to Figure 10. But we have one more variation to try: $\hat{\hat{E}}'' = \{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_5\}), \chi\{(\mathcal{A}_3, \mathcal{A}_4\})\}$ (Figure 13)



(a) $\chi(\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4\})$ (b) $\mathcal{A}_5$

Figure 9: Five players passing around a ball: $E''' = \{\chi(\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \}), \mathcal{A}_5\}$

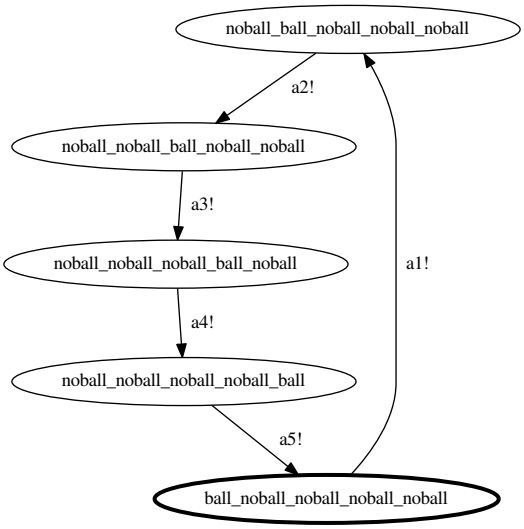
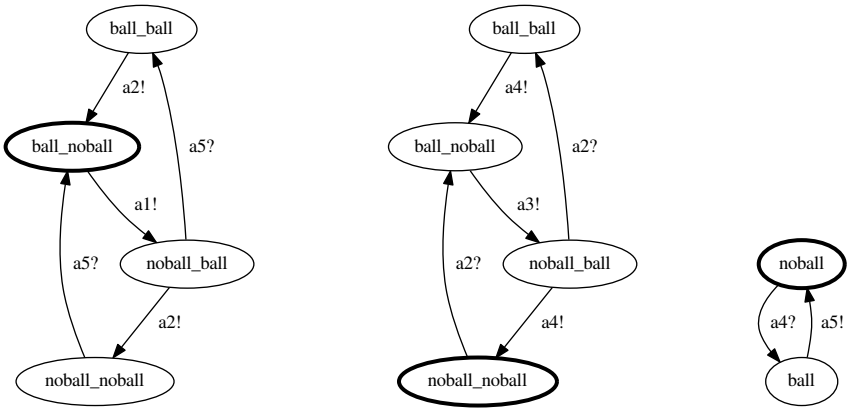
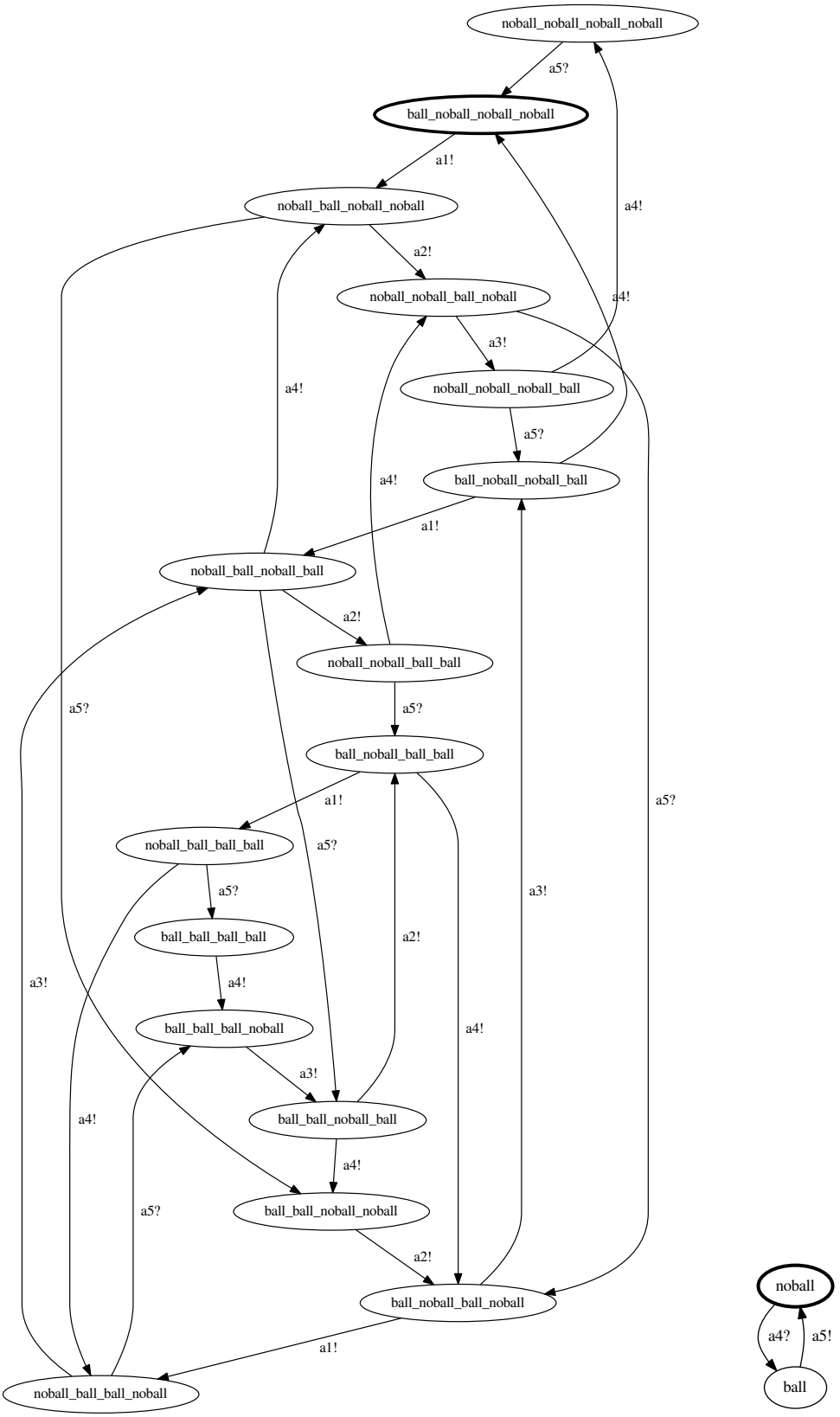


Figure 10: Five players passing around a ball: $\chi(\{\chi(\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \}), \mathcal{A}_5\})$



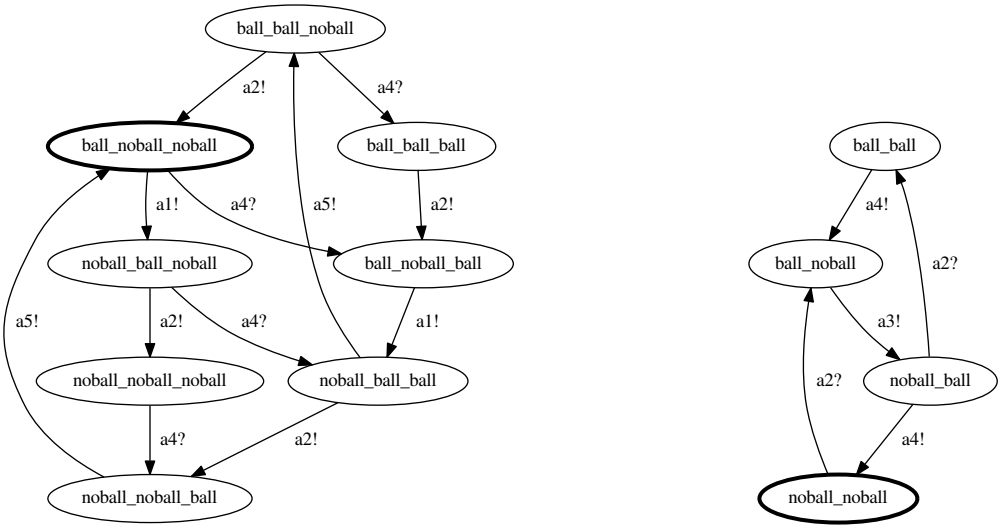
(a) $\chi(\{\mathcal{A}_1, \mathcal{A}_2\})$ (b) $\chi(\{\mathcal{A}_3, \mathcal{A}_4\})$ (c) $\mathcal{A}_5$

Figure 11: Five players passing around a ball: $\hat{E}'' = \{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \chi(\{\mathcal{A}_3, \mathcal{A}_4\}), \mathcal{A}_5, \}$



(a) $\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \chi(\{\mathcal{A}_3, \mathcal{A}_4\})\})$ (b) $\mathcal{A}_3$

Figure 12: Five players passing around a ball: $\hat{E}'' = \{\chi(\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \chi(\{\mathcal{A}_4, \mathcal{A}_5\})\}), \mathcal{A}_3\}$



(a) $\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_5\})$

(b) $\chi(\{\mathcal{A}_3, \mathcal{A}_4\})$

Figure 13: Five players passing around a ball: $\hat{\hat{E}}'' = \{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_5\}), \chi\{(\mathcal{A}_3, \mathcal{A}_4\})\}$

So what is the use of all these different ways of nesting an automaton? Is one method somehow better? We ran these through the Python implementation (see below) which records the effort performed at each synchronization. "Effort" is the amount of tests to see if component $\mathcal{A}_i$ is enabled for action a at state p performed by Algorithm 3, as discussed in section 4.4.

nesting	**effort**	**picture**
$\{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5\}$	50	Figure 6
$\{\chi(\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_3\}), \mathcal{A}_4, \}), \mathcal{A}_5\}$	283	Figure 9
$\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \chi(\{\mathcal{A}_3, \mathcal{A}_4\}), \mathcal{A}_5\}$	163	Figure 12
$\{\chi(\{\chi(\{\mathcal{A}_1, \mathcal{A}_2\}), \mathcal{A}_5\}), \chi\{(\mathcal{A}_3, \mathcal{A}_4\})\}$	107	Figure 13

Clearly, the uncomplicated "all at once" approach did well. But even for such a simple system, the other approaches show a significant difference in performance. In the Experiments section we will experiment with different nestings of sub-products.

5 Software

We have written a Python 2.7 implementation using the results from the earlier sections. We chose Python as a programming language because it offers all the features we need to implement the algorithms in a natural way: object orientation; good data structures for tuples, sets, lists, dictionaries and strings; imperative and functional programming capabilities; good interface with Graphviz for visual rendering of results. Aside from that, Python is widely available and easy to read. We chose object-orientation as a programming paradigm because it is a natural fit for the very object-like team automata.

The software assumes deterministic automata with finite sets of states.

The software starts with the TA class (in `TA.py`), which implements team automata as objects having a name, sets of internal, input and output actions, states, transitions and an initial state as attributes. This class also provides various member functions for manipulating it, such as adding or removing transitions, checking for livelock, generating a corresponding RRTS and so forth. Of particular use is the `screendump()` function which prints a human-readable rendering of the automaton to the screen. Apart from those, the TA class also has two sets that start out empty: `message_losses` and `deadlocks`. These are used by the synchronization algorithm to record any *directly incompatible* states encountered.

The TA module also governs the file I/O of automata to `.out` files. The file format is intended to be compatible with an earlier software project by Casanova (see 5.1), and that software uses the `.out` suffix on output files.

TAs are stored in text files, which look like this:

```
.model drinker
.inputs serve_coffee
.outputs press_button
.internal drink_coffee
.state graph
    thirsty press_button waiting
    waiting serve_coffee happy
    happy drink_coffee thirsty
.marking {thirsty}
.end
```

../source_code/cases/drinker.out

Model, inputs, outputs, internal, state graph, marking and end are reserved words. The marking is a set containing one state. Only standard alphabet characters and underscore are permitted, and whitespace is a delimiter. It is not exactly user-friendly but necessary to maintain compatibility with the software of Casanova. While interacting with our software, we can abstract most of this to a more pleasant format.

The next module is `synchronize.py` which contains our `synchronous_product_effort(components = [], find_all_contradictions=True)` function. This function computes the synchronous product of the list of components given, as well as the amount of effort required. These are then returned as a `(result, effort)` tuple. If any incompatibilities were encountered, these will be found in `result.message_losses` or `result.deadlocks`. If both sets remain empty then the components were compatible.

The actual working of `synchronous_product_effort` is based on Algorithm 3; starting in the initial state it explores a known-reachable state, checks it for direct incompatibility, determines what transitions are enabled, adds those transitions and their destinations to `result`, and goes on to explore any other states discovered to be reachable but as yet unexplored, until all reachable states have been explored.

`synchronous_product_effort` by default does not stop if it discovers a directly incompatible state. Rather, it records it and goes on to explore the remaining reachable states. In this way, when the algorithm concludes it has computed *DIS* so that a developer can use it to debug. By giving it `find_all_contradictions=False` as an argument this is changed so that it aborts a synchronous product if it finds a contradiction.

Our next layer of programming is `scheduler.py`. This module provides a `scheduler` class which processes schedule files. It first recursively calls schedulers to resolve any sub-schedules (nested synchronous products), before taking on the system of automata to test for compatibility.

The scheduler will process the system by selecting two components and taking their synchronous product to see if they are compatible. If they are, then (justified by Lemma 30) it removes those two components from the system and replaces them with their product. If they are not component, the product is discarded and a different pair is selected from a queue of eligible pairs. When no more eligible pairs remain, all remaining components are sent to `synchronous_product_effort` together to

see if they are compatible. If they are, the system was compatible and the result is a TA corresponding (via Lemma 28 and Lemma 29) to the synchronous product of the original system. Otherwise, the system was not compatible, but the corresponding product is still produced along with its sets of message loss and deadlock states, for debugging purposes.

The schedule file is a text file containing one line describing the system, and looks like this:

$$(a1, a2, (a3, a4))$$

Which corresponds to $\{\mathcal{A}_1, \mathcal{A}_2, \chi\rho(\{\mathcal{A}_3, \mathcal{A}_4\})\}$. When executed by the scheduler, it will first run another scheduler to replace (a3, a4) with their synchronous product, without caring about their compatibility. Then it will proceed to test the compatibility of (a1, a2, Xa3a4).

As mentioned above, the scheduler maintains a queue of eligible pairs for nested synchronization. This queue is formed by, for every possible pair, computing a fitness score. When the time comes to try a new pair, the fittest pair that satisfies the output collaboration requirement of Lemma 30 is used. If it is compatible, all other pairs that overlap with it are dropped and the result of the synchronization is compared with all remaining components to evaluate the fitness of those new possible pairs.

So how is fitness computed? The scheduler uses the `analyze(A, B)` function as an alias for a strategy function. If a strategy argument was given to the scheduler, any calls to `analyze(A, B)` are actually executed by a matching `strategy_[strategyname](A, B)` function instead which records the fitness of (A, B), if appropriate.

Built-in strategies include assigning high fitness to pairs which have relatively many shared actions, FIFO and LIFO, and pairs which could result in large or small $|Q|$ for their synchronous product. This part of the module is intended to be easy to expand with further strategies that come to mind.

When the scheduler class constructor is called, its parameter `self.contradictions` is set. If it is set to True, then the scheduler will use `synchronous_product_effort` with the `find_all_contradictions` parameter set to True; and False if `self.contradictions` is False.

Also, components loaded from file are first reduced. After reduction components are checked for livelocks. Justified by Lemma 15, if `self.contradictions` is set to False, if it detects a livelock the scheduler will quit immediately. If set to True, the livelock in the initial component will be reported but otherwise allowed. Any products of synchronization that retain those livelocks are still considered incompatible however, so they will not be used. If a livelock should happen to be pruned away in the (reduced!) synchronous product, then the final result may still be compatible.

A separate command-line module is available to use the scheduler. It is called `scheduler_commandline.py` and accessed from the `/source_code/cases` directory by calling `python ../scheduler_commandline [strategy=optional strategy argument] [complete=optional completeness argument] schedule=[mandatory schedule name]`. If no strategy is chosen it will use FIFO. If completeness is not specified then `self.contradictions` in the scheduler class instance will be set to False. To set it to true use `complete=True` as an argument.

We can produce a visualization of an automaton with the `graph.py` module. It contains the `makeGraph(subject = TA.TA())` function, which takes a single TA object as an argumentt. This module can also be accessed from a commandline with `graph_commandline.py` using the name of the automaton file as an argument. An optional argument can be given to specify the file format, otherwise the format is Scalable Vector Graphics (SVG).

The programs allow for automated experimentation; a simple example of this can be found in the `experiment.py` file.

Files, contents:

- **TA.py** – the team automaton class.
- **synchronize.py** – function to compute the synchronous product of a set of n team automata.
- **scheduler.py** – runs nested compatibility testing according to a strategy.
- **scheduler_commandline.py** – run a schedule from the command line.

  Called from `/source_code/cases` as:
  `python ../scheduler_commandline [strategy=optional strategy argument] [complete=optional completeness argument] schedule=[mandatory schedule name]`
- **graph.py** – makes graphical output.

- **graph_commandline.py** – command-line interface for `graph.py`.

  Called from `/source_code/cases` as:
  ```
  python ../graph_commandline [optional file format argument]
  [mandatory filename argument]
  ```

- **experiment.py** – example use of the other modules to test a collection of schedules of automata at the press of a button.

  Called from `/source_code/cases` as:
  ```
  python ../experiment.py
  ```

Dependencies:

- Python 2.7 modules:
  - functools
  - graphviz
  - itertools
  - math
  - operator
  - os
  - pyparsing
  - random
- Graphviz

Tested on Ubuntu 14.04 LTS

5.1 Other software

As far as we are aware, only one earlier software implementation of team automata exists, written by Jonas Casanova and found on `sourcefourge.net` under the name **synchroteam**. [3] It is available under the GNU General Public License version 2.0 (GPLv2).

It computes products of automata; not only synchronous products but it also allows the specification of other synchronization types. It does not however determine compatibility. We have aimed to maintain compatibility with synchroteam by using the same file format for components.

We were inspired by its use of Graphviz to automatically produce visualizations from component files, and adapted the technique to use Python's Graphviz module.

6 Experiments

We have done some initial experiments with our Python implementation, to get a first impression of the effect of nested compatibility checking using different strategies. We used `experiment.py` to run our cases through `scheduler.py` using strategies detailed below. In all cases did we use the same setting for our sychronous product/compatibility checking function `synchronous_product_effort(components = [], find_all_contradictions=True)`: we set it to stop as soon as the first contradiction was found.

We used the following strategies:

- **action similarity:** $fitness = \frac{|\Sigma_1 \cap \Sigma_2|}{|\Sigma_1 \cup \Sigma_2|}$

  The idea being that components that share relatively many actions are important to each other and that it is therefore a good idea to combine those first.

- **action dissimilarity:** $fitness = \frac{-|\Sigma_1 \cap \Sigma_2|}{|\Sigma_1 \cup \Sigma_2|}$

  This strategy was inspired as the opposite of action similarity, to see if anything interesting happens.

- **first in first out:** $fitness = last\ assigned\ fitness - 1$

  The oldest pair is always the favourite for synchronization testing. All original components will be used (if possible) before any products of previous synchronizations are used.

  In the passing the ball example, this would have resulted in a series of Figures 6, 7, 11, 13, 10.

- **last in first out:** $fitness = last\ assigned\ fitness + 1$

  Newly produced components will always be favourite to be paired with second-youngest components. This corresponds to Figures 6–10.

- **Q minimization:** $fitness = -|Q_1| \times |Q_2|$

  The idea being that we want to keep our components as small as possible so that synchronizing them is algorithmically cheaper.

  In the passing the ball example, this would have resulted in a series of Figures 6, 7, 11, 13, 10.

- **Q maximization:** $fitness = |Q_1| \times |Q_2|$

  Again an anti-strategy, to see if there are situations where going for the big components is better.

  This corresponds to Figures 6–10.

- **all at once:** synchronize all components together at once, instead of pairwise. The goal of all the other strategies is to beat this one.

  This corresponds to Figure 6 going directly going to Figure 10.

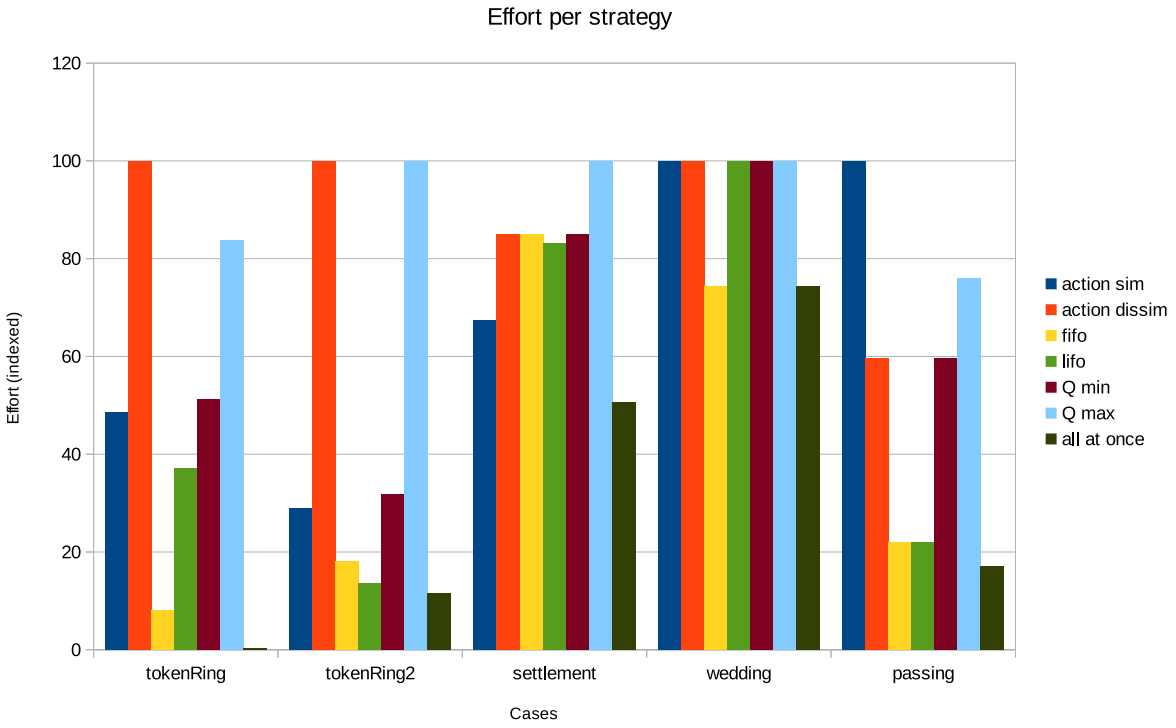
It should be noted that with *action (dis)similarity* and *Q minimization/maximization* multiple pairs might get the same fitness score. In that case the program breaks ties in a nondeterministic manner.

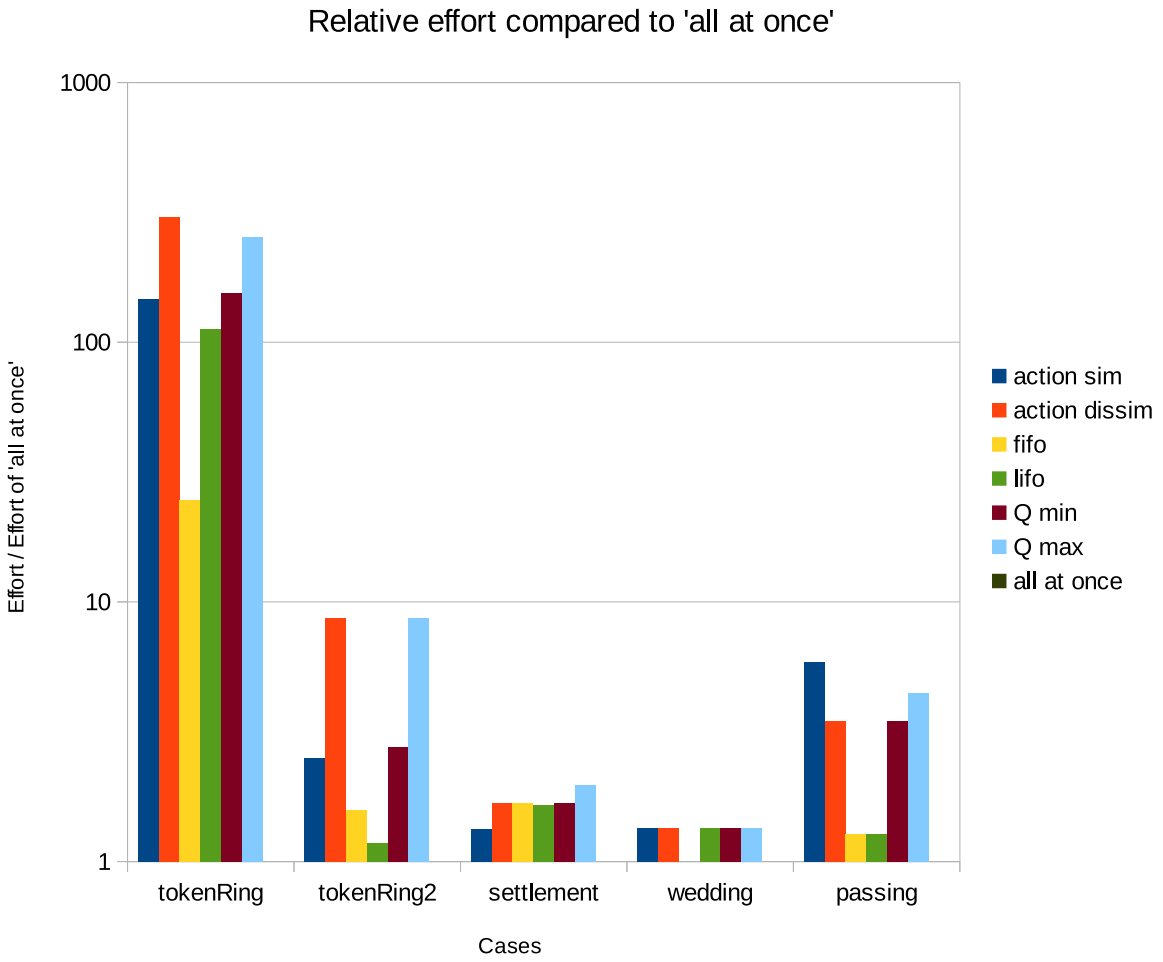
The cases we used were the components already mentioned earlier (coffee machine/drinker, workplace, passing the ball) as well as a wedding (using the groom, but with his livelock removed), a token ring system that was also used as an example in [2] as well as an abbreviated version of it (tokenRing2), and a settlement process inspired by [6]. All of these are included as component specifications in the appendix.

The results are as follows:

	Compatible?	action sim	action dissim	fifo	lifo	Q min	Q max	all at once
workplace	False	4	4	4	4	4	4	4
coffeeTime	False	20	20	20	20	20	20	20
tokenRing	False	2500	5157	421	1916	2637	4324	17
tokenRing2	True	633	2185	397	297	696	2185	252
settlement	False	176	222	222	217	222	261	132
wedding	True	430	430	320	430	430	430	320
passing	True	292	174	64	64	174	222	50

The results for “workplace” and “coffeeTime” are trivial: when a system contains only two components every strategy will achieve the same score.





Several facts immediately jump out. Clearly, no nested compatibility checking strategy was able to beat the default method of checking all components together. The strategies *action dissimilarity* and *Q max* consistently performed poorly.

FIFO did surprisingly well, notably outperforming *Q minimization*. This could be a random result; the performance of *FIFO* and *LIFO* is influenced by the order in which components are presented in the original schedule. (A1, A2, A3, A4) could score differently than (A3, A1, A2, A4). Still, it is plausible that *FIFO* is simply a good strategy. Teams have a tendency to experience exponential growth, so making many few-generation products should work better than one many-generation product that keeps adding components to its output. Which is why the occasionally good performance of *LIFO* is notable. An explanation may be that schedules tend to be somewhat organized: a developer will list related components together.

Another significant finding is that no single strategy was the best one every time. This leads to the idea that perhaps different strategies are ideal for different kinds of components. A system where many components do not share actions with more than a few other components might benefit more from *action similarity* perhaps. If components tend to have spare transitions between their states, perhaps a different strategy dominates than if they are internally well-connected. And a system which is supposed to follow a single path from start to finish may benefit from a different strategy than one that branches into different paths.

In any case, we should ask ourselves whether there is any point to these strategies given that "all at once" outperformed all of them. An answer may be that Yes, there is a point: distribution. The all-at-once approach is far harder to implement in a distributed manner than some of these strategies. However, to pick the right strategy, a better insight is required in how to classify styles of team automata so that the best heuristic can be picked for nesting the work.

7 Conclusions

We have succeeded at our primary goal, to find and implement an algorithm that determines whether a system of n components is *compatible*. It meets the criteria we established: it delivers a result, consisting of a compatible/not-compatible verdict, as well as an actual team over the system.

Our methods maintain the integrity of the original input with regards to current and future behaviour; to an outside observer our reduced system appears to behave exactly the same as the original system would have. The only things we remove are unreachable states, useless transitions and useless internal actions. To ensure that the team has the same effect in future systems, we maintain useless input and output actions. In all cases, our results remain consistent with the basis inherited from [2], so that all our results are applicable in an environment according to their criteria.

While producing an algorithm was the primary goal, making sure it was a useful algorithm was of course also a priority. Analysis of its complexity revealed it to be exponential. So we sought to find improvements, to try to reduce its exponential complexity. We considered an alternative algorithm which turned out more unfeasible. We examined pruning methods and a divide and conquer approach. None of these succeeded in reducing the complexity to a lower order. It seems likely to us that this is simply not possible – finding out if states with certain properties are reachable closely resembles the question of computing the intersection of the languages of two finite automata, which is known to be an NP-complete problem ([5], page 266). So the question turns to finding ways to make the best of a bad situation. And it turns out there are some promising leads there.

We have been able to reduce the requirements for using a nested approach to compatibility checking and generalize the proof that it is possible. This makes it theoretically possible to split up a compatibility checking task into multiple smaller tasks so that we can distribute the workload, taking advantage of concurrency.

However, this requires a strategy to select an order in which to try subteams. We have demonstrated that several strategies are possible. The strategies we implemented are by no means the only possible ones however; we have provided a framework to which new strategies can be added. An interesting finding is that no nesting strategy consistently outperformed all the others. This suggests that we could discover which strategies work best in which situations.

7.1 Future Questions

A major open question is how useful nested checking is, considering the much better performance of checking all components together at once. Determining if the non-nested checking forms an upper bound on the performance of nested strategies would be worthwhile.

A topic related to nested checking is the lack of a typology of team automata. The model can represent many different things, but we have no standard of what to expect. We cannot say that it is normal for a component to have states sparsely or densely connected with transitions, or to have large or small alphabets. But we could develop a typology to identify certain categories of components which have typical combinations of such properties. The value here is that we can then work on picking a strategy that best matches the system we are working on.

A hindrance when testing team automata is the lack of a major test set. We looked at the possibility of generating a test set with examples of various properties. We believe this is in fact a very interesting avenue for future research because it posits the quesion: how do we generate an automaton that will have a property X that we desire – such as compatibility!

Since checking for incompatibility turns out to be so costly, developing methods to write models guaranteed to be compatible is perhaps a more rewarding route to take. Lemma 30 hints at where this may lead: developers may be instructed to ensure that the component they are responsible for will collaborate on certain actions, be always receptive to other actions etcetera. Obviously, models which guarantee collaboration on output are of interest here.

An entirely different direction of research would be to examine how well the notion of compatibility serves us in practice. We examined the use of team automata as applied to a project of Pieter Kwantes e.a. in [6] to translate Bussiness Process Modeling Notation diagrams into statements that can be logically checked. At first, team automata appeared like an obvious tool for this, but translating the compliance requirements was rather an ad-hoc process. It was possible to model failure to follow a settlement protocol as a failure of compatibility, but this required quite a bit of engineering the model.

Related to that, we have some lingering doubts about the definition of livelock. There is a discrepancy between the definitions for livelock used in [2] and [1]; the latter is only concerned with reachable

livelocks. But is this really the sum of possible livelocks? Consider a system where two components need to both be ready before they can execute an action that the rest of the system needs in order to progress. However, component 1 is ready and component 2 is not. On the clock signal (an input for both), the ready component becomes unready and the unready component becomes ready. This system could still meet all the criteria for compatibility, but it looks like a two-component livelock in all but name.

References

[1] Josep Carmona and Jordi Cortadella. Input/output compatibility of reactive systems. In *Fourth International Conference on Formal Methods in Computer-Aided Design*, pages 360–377, 2002.

[2] Josep Carmona and Jetty Kleijn. Compatibility in a multi-component environment. *Theoretical Computer Science*, 484:1–15, 2013.

[3] Jonas Casanova. Synchroteam: team automata synchronizer. http://synchroteam.sourceforge.net/, 2013. [Online; accessed 28-August-2015].

[4] Clarence A. Ellis. Team automata for groupware systems. In *GROUP '97 Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, pages 415–424, 1997.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[6] Pieter M. Kwantes, Pieter Van Gorp, Jetty Kleijn, and Arend Rensink. Towards compliance verification between global and local process models. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *Graph Transformation*, volume 9151 of *Lecture Notes in Computer Science*, pages 221–236. Springer International Publishing, 2015.

[7] Maurice H. ter Beek, Clarence A. Ellis, Jetty Kleijn, and Grzegorz Rozenberg. Synchronizations in team automata for groupware systems. *Computer Supported Cooperative Work*, 12:21–69, 2003.

Index

action-complementary, **12**, 14, 27
actions, 7, **8**
active behaviour, **24**, 24
active behaviour of a component in a team is a subset of its solitary behaviour, 24
active computation, **23**, 23
active computations, 23
alphabet, **7**, 7, 8

behaviour, 5, **23**, 23

collaborates, **12**, 12, 14, 27
collaborates on output, **28**
communicating, **9**
compatibility, 10, **11**, 11, 19
compatibility relation, **11**, 21
compatible, **12**, 12, 14, 19, 21, 28, 45
complete, **9**, 14
complete transition space, **9**
component's active behaviour in a team is a subset of active behaviour in subteams containing it, 24
composable system, **9**, 9, 11, 15
computation, **8**, 8
computations, 23
correspondence of compatibility, 28
correspondence of states, 27
correspondence of transitions, 27

deadlock-free, **12**, 12, 14, 28
Definitions
  active behaviour, 24
  active computation, 23
  alphabet, 7
  behaviour, 23
  compatibility, 11
  composable system, 9
  computation, 8
  determinism, 17
  directly incompatible state, 19
  projection, 7
  synchronous product, 10
  team automata, 9
  transition system, 8
  ultimately incompatible state, 21
  usefulness, 23
determinism, 17
deterministic, **17**
directly incompatible, **19**, 19, 37
directly incompatible state, 19
DIS, **19**, 21, 37
domain, **9**
domain enabled, **9**

enabled, **8**

I not in UDIS, 21

input domain, **9**
input-domain enabled, **9**
input-enabled, **8**

Lemmas
  correspondence of compatibility, 28
  correspondence of states, 27
  correspondence of transitions, 27
  I not in UDIS, 21
  persistence of livelock-freeness, 17
livelock, **11**
livelock-free, **11**, 12

nondeterministic, **17**

output domain, **9**
output-domain enabled, **9**

persistence of livelock-freeness, 17
progressive, **12**
projection, **7**, 7
propagation of uselessness, 24

reachable, **8**, 19, 24
reactive, **8**
reactive transition system, 7, **8**, 15
receptive, **12**, 12, 14, 28
receptiveness, 12
Reduced Reactive Transition System, **26**
RRTS, **26**
RTS, **8**, 11, 26

states, **8**
states in the compatibility relation are not in UDIS, 21
synchronization, **10**
synchronizations, 9
synchronous product, **10**, 10–12, 15, 24, 25
synchroteam, **39**

TA, **9**
team automata, 9
team automaton, 7, 8, **9**
terminating, **11**
Theorems
  active behaviour of a component in a team is a subset of its solitary behaviour, 24
  component's active behaviour in a team is a subset of active behaviour in subteams containing it, 24
  propagation of uselessness, 24
  states in the compatibility relation are not in UDIS, 21
transition system, 7, **8**, 8
transitions, 7, **8**
TS, **8**, 17, 23

UDIS, **21**, 21

ultimately incompatible, **21**
ultimately incompatible state, 21
unreachable, **8**
useful, **23**, 26
usefulness, 23
useless, **23**, 23

A Code

```
#! /usr/bin/env python

class TA:

    def __init__(self, filename=None):
        """Initialize from file or blank (no argument)

        File should be formatted thus:

        .model MODELNAME
        .inputs INPUT1 INPUT2
        .outputs OUTPUT1 OUTPUT2
        .internal INTERNAL1 INTERNAL2
        .state graph
        CURRENTSTATE1 ACTION1 NEXTSTATE1
        CURRENTSTATE2 ACTION2 NEXTSTATE2
        .marking INITIALSTATE
        .end

        CAPITALS indicate variables. lowercase indicates keywords.
        """
        # create blank TA
        if filename is None:
            self.name = "Anon"
            self.states = set()
            self.input_actions = set()
            self.output_actions = set()
            self.internal_actions = set()
            self.initial_state = ""
            self.transitions = {}  # (current state, action) : next state
            self.message_losses = set()
            self.deadlocks = set()
            return
        # create TA from file
        f = open(filename, "rb")
        reading_transitions = False
        self.transitions = {}
        self.states = set()
        self.message_losses = set()
        self.deadlocks = set()
        for line in f:
            if ".model" in line:
                self.name = line.strip().split(" ")[1]
            elif ".inputs" in line:
                self.input_actions = set(line.strip().split(" ")[1:])
            elif ".outputs" in line:
                self.output_actions = set(line.strip().split(" ")[1:])
            elif ".internal" in line:
                self.internal_actions = set(line.strip().split(" ")[1:])
            elif ".state graph" in line:
                reading_transitions = True
            elif ".marking" in line:
                reading_transitions = False
                temp = line.strip().split("{")[1].split("}")[0]
                if " " in temp:
                    print "Error: multiple initial states."
                    print line
                self.initial_state = temp
            elif ".end" in line:
                break
            elif reading_transitions == True:
                temp = line.strip().split(" ")
                if len(temp) != 3:
                    print "Error: weird transition:"
                    print line
                if (temp[0], temp[1]) in self.transitions:
                    print "Error: nondeterministic transition:"
                    print line
                else:
                    self.add_transition(temp[0], temp[1], temp[2])
```

```
72      def get_all_actions(self):
            """Return the union of input, output and internal alphabets."""
74          return self.output_actions.union(self.input_actions).\
                union(self.internal_actions)
76
        def sanity_check(self):
78          """Check wether a TA is properly initialized and whether it conforms
            to the requirements for composability.
80
            TA sanity requirements:
82              - It must have a nonempty initial state. This implementation uses
                deterministic TAs, so exactly one initial state is required.
84              - The initial state must belong to the overall set of states.
                - Transitions go from states in Q to other states in Q.
86              - Transitions use an action from the Input, Output or Internal
                alphabet.
88              - Input, Output and Internal alphabets cannot overlap.

90          Note that this function does NOT check for livelock-freeness.
            """
92          if self.initial_state == "":
                print "Initial state not set."
94              return False
            if self.initial_state not in self.states:
96              print "Initial state not in Q."
                return False
98          if not self.input_actions.isdisjoint(self.output_actions):
                print "Input and output actions overlap"
100             return False
            if not self.input_actions.isdisjoint(self.internal_actions):
102             print "Input and internal actions overlap"
                return False
104         if not self.output_actions.isdisjoint(self.internal_actions):
                print "Output and internal actions overlap"
106             return False
            for item in self.transitions:
108             if item[0] not in self.states:
                    print "Bad transition ", item, self.transitions[item],
110                 print "; starting state not in Q."
                    return False
112             if ((item[1] not in self.input_actions) and
                    (item[1] not in self.output_actions) and
114                 (item[1] not in self.internal_actions)):
                        print "Bad transition ", item, self.transitions[item],
116                     print "; action not in Sigma."
                        return False
118             if self.transitions[item] not in self.states:
                    print "Bad transition ", item, self.transitions[item],
120                 print "; destination state not in Q."
                    return False
122         return True # no insanity detected

124     def screendump(self):
            """Dump the automaton to the screen.
126         """
            print "Name: " + self.name
128         print "    states:"
            for item in self.states:
130             print "        " + item
            print "    input actions:"
132         for item in self.input_actions:
                print "        " + item
134         print "    output actions:"
            for item in self.output_actions:
136             print "        " + item
            print "    internal actions:"
138         for item in self.internal_actions:
                print "        " + item
140         print "    initial state: " + self.initial_state
            print "    transitions:"
142         for item in self.transitions:
                print ("        (" + item[0] + ", " + item[1] + ", " +
144                 self.transitions[item] + ")")
```

```
        if len(self.message_losses) > 0:
            print "    MESSAGE LOSSES:"
            for item in self.message_losses:
                print "        " + item[0] + " > " + item[1]
        if len(self.deadlocks) > 0:
            print "    DEADLOCKS:"
            for item in self.deadlocks:
                print "        " + item

    def save(self, filename=None):
        """Writes the automaton to file, using the fileformat from Synchroteam

        Insists on ".out" as file extension; appending if necessary.
        If no filename is given, the automaton name is used as filename.

        For more information about Synchroteam:
        http://sourceforge.net/projects/synchroteam/
        """
        if filename == None:
            filename = self.name + ".out"
        elif not filename.endswith(".out"):
            filename = filename + ".out"
        with open(filename, "wb") as d:
            d.write(".model " + self.name +"\r\n")
            d.write(".inputs")
            for item in self.input_actions:
                d.write(" " + item)
            d.write("\r\n")
            d.write(".outputs")
            for item in self.output_actions:
                d.write(" " + item)
            d.write("\r\n")
            d.write(".internal")
            for item in self.internal_actions:
                d.write(" " + item)
            d.write("\r\n")
            d.write(".state graph\r\n")
            for item in self.transitions:
                d.write("    " + item[0] + " " + item[1] + " " + self.transitions[item])
                d.write("\r\n")
            d.write(".marking {" + self.initial_state + "}\r\n")
            d.write(".end\r\n")

    def rename_internal_actions(self):
        """Renames internal actions by prepending TA name.

        In a composable systems, component automata are not allowed to share
        internal actions. This method aims to ensure that all internal actions
        receive a unique name.
        """
        temp = set()
        for action in self.internal_actions:
            temp.add(self.name + "_" + action)
        self.internal_actions = temp

    def is_action_enabled(self, state, action):
        """Returns true if the action is enabled, false if not.

        Action is enabled if (current state, action, next state) IN transitions
        """
        if (state, action) in self.transitions:
            return True
        else:
            return False

    def get_enabled_actions(self, state):
        """Get all actions that are enabled in the given state.
        """
        return {x for x in self.get_all_actions() if self.is_action_enabled(state, x)}

    def get_next_state(self, state, action):
        """Get the next state if the given action is taken in the given state.
        """
```

```
218         if (state, action) in self.transitions:
220             return self.transitions[(state, action)]
            else:
222             print "ERROR:", state, action, "not defined."
                return None

224     def get_directly_reachable_states(self, state):
            """Get the states that can be directly reached from the current state
226         """
            return {x for x in self.states if x in {self.get_next_state(state, y)
228             for y in self.get_enabled_actions(state)}}

230     def get_reachable_states(self):
            """Get all states that can be reached eventually
232
            Starting in the initial state, add the directly reachable states of
234         known reachable states until the set stops growing.
            """
236         result = {self.initial_state}
            todo = self.get_directly_reachable_states(self.initial_state)
238         while len(todo) != 0:
                current = todo.pop()
240             result.add(current)
                todo |= {x for x in self.get_directly_reachable_states(current) if
242                 x not in result}
            return result
244
        def get_useful_transitions(self):
246         """Returns the set of all useful transitions.

248         A transition is useful if it starts in a reachable state.
            """
250         return {(x[0], x[1], self.transitions[x]) for x in self.transitions if
                x[0] in self.get_reachable_states()}
252
        def get_useful_actions(self):
254         """Returns the set of all useful actions.

256         An action is useful if it is used by at least one useful transition.
            """
258         return {x[1] for x in self.get_useful_transitions()}

260     def get_unreachable_states(self):
            """Returns all unreachable states.
262         """
            return {x for x in a.states if x not in a.get_reachable_states()}
264
        def get_useless_transitions(self):
266         """Returns all useless transitions.
            """
268         return {(x[0], x[1], self.transitions[x]) for x in self.transitions if
                x[0] not in self.get_reachable_states()}
270
        def get_useless_actions(self):
272         """Returns all useless actions.
            """
274         return {x[1] for x in self.get_all_actions() if x not in
                self.get_useful_transitions()}
276
        def add_state(self, state):
278         """Add a new state to the TA.
            """
280         if isinstance(state, str):
                self.states.add(state)
282         else:
                print "Error: adding a non-string state"
284
        def add_internal_action(self, action):
286         """Adds a new internal action to the TA.
            """
288         if isinstance(action, str):
                if (action in self.input_actions):
290                 print "Error: action is already an input action."
```

```
            elif (action in self.output_actions):
                print "Error: action is already an output action."
            else:
                self.internal_actions.add(action)
        else:
            print "Error: adding a non-string action"

    def add_input_action(self, action):
        """Adds a new input action to the TA.
        """
        if isinstance(action, str):
            if (action in self.internal_actions):
                print "Error: action is already an internal action."
            elif (action in self.output_actions):
                print "Error: action is already an output action."
            else:
                self.input_actions.add(action)
        else:
            print "Error: adding a non-string action"

    def add_output_action(self, action):
        """Adds a new output action to the TA.
        """
        if isinstance(action, str):
            if (action in self.internal_actions):
                print "Error: action is already an internal action."
            elif (action in self.input_actions):
                print "Error: action is already an input action."
            else:
                self.output_actions.add(action)
        else:
            print "Error: adding a non-string action"

    def add_transition(self, start_state, action, end_state):
        """Adds a new transition to the TA.

        If start/end state not yet defined, will also be added.
        Actions cannot be implicitly added because we'd lack information on
        whether they're input/output/internal.
        """
        self.add_state(start_state)
        self.add_state(end_state)
        if action not in self.get_all_actions():
            print "Error: action not defined:", action
        else:
            self.transitions[(start_state, action)] = end_state

    def set_initial_state(self, state):
        """Set the initial state to a new value.
        """
        if isinstance(state, str):
            if (state in self.states):
                self.initial_state = state
            else:
                print "Error: suggested new initial state not in set of states."
        else:
            print "Error: new initial state not a string."

    def remove_state(self, state):
        """Remove a state from the TA's set of states.

        It is an error to remove the initial state.
        When a state is removed, all transitions starting or ending
        in that state are also removed.
        """
        if state in self.states:
            if state is self.initial_state:
                print "Error: trying to remove initial state."
            else:
                self.states.remove(state)
                for t in self.get_enabled_actions(state):
                    del self.transitions[(state, t)]
                for t in {t for t in self.transitions if self.transitions[t]
```

```
                    is state}:
                        del self.transitions[t]
        else:
            print "Error: trying to remove nonexistent state."

    def remove_transition(self, start_state, action):
        """Remove a transition.
        """
        if (start_state, action) in self.transitions:
            del self.transitions[(start_state, action)]
        else:
            print "Error: that transition doesn't exist."

    def remove_action(self, action):
        """Remove an action from the alphabet.

        All transitions using that action will also be removed.
        """
        if action in self.internal_actions:
            self.internal_actions.remove(action)
            for t in {t for t in self.transitions if t[1] is action}:
                del self.transitions[t]
        elif action in self.input_actions:
            self.input_actions.remove(action)
            for t in {t for t in self.transitions if t[1] is action}:
                del self.transitions[t]
        elif action in self.output_actions:
            self.output_actions.remove(action)
            for t in {t for t in self.transitions if t[1] is action}:
                del self.transitions[t]
        else:
            print "Error: cannot remove nonexistent action"

    def reduce_this_automaton(self):
        """Reduces this automaton.

        Removes all unreachable states, which will also remove all useless
        transitions; useless actions remain.
        """
        for s in self.get_unreachable_states():
            self.remove_state(s)

    def get_enabled_internal_actions(self, state):
        """Get all internal actions that are enabled in the given state.
        """
        return {x for x in self.internal_actions if self.is_action_enabled(
            state, x)}

    def get_internal_next_states(self, state):
        """Get all states that can be directly reached by an internal action.

        Intended for livelock checking.
        """
        return {x for x in self.states if x in {self.get_next_state(state, y)
            for y in self.get_enabled_internal_actions(state)}}

    def detect_livelock(self, quiet=False):
        """Find livelocks anywhere in the automaton.

        Livelocks are found by selecting states with an outgoing internal
        transition, then for each of them calling a graph exploration function
        that only travels along internal transitions.

        Note that this function can also detect unreachable livelocks if the
        automaton contains unreachable states. It is recommended to use this
        function on automata already reduced with the reduce_automaton or
        get_reduced_automaton functions.
        """
        safe = set()
        todo = {x for x in self.states if
            len(self.get_enabled_internal_actions(x)) > 0}
        for item in todo.difference(safe):
            safe, live = self.explore_internal_paths(safe, so_far = [item], quiet=quiet)
```

```
            if live:
438             return True
        return False
440
    def explore_internal_paths(self, safe, so_far, quiet=False):
442     """Explore internal paths, to detect livelock

444     So_far is the states already reached. If a possible next state is also
        in so_far, that means a livelock exists, and the function returns.
446     Otherwise it recurses on next states until it finds a longer livelock
        or it exhausts all possibilities.
448     """
        for x in self.get_internal_next_states(so_far[-1]).difference(safe):
450         if x in so_far:
                if quiet==False:
452                 print "Livelock found:", so_far, x
                return (safe, True)
454         else:
                so_far.append(x)
456             safe, live = self.explore_internal_paths(safe, so_far, quiet)
                if live:
458                 return (safe, True)
                so_far.pop()
460     safe.add(so_far[-1])
        return (safe, False)
462
    def get_reduced_version(self):
464     """Returns a reduced version of this automaton.

466     The reduced version will contain:
            - States IFF they are reachable
468         - Transition IFF they are useful
            - Internal actions IFF they are useful
470         - All Input and Output actions
        """
472     RRTS = TA()
        RRTS.name = "R_" + self.name
474     RRTS.initial_state = self.initial_state
        RRTS.input_actions = self.input_actions
476     RRTS.output_actions = self.output_actions
        todo = [self.initial_state]
478     done = set()
        while len(todo) != 0:
480         for action in self.get_enabled_actions(todo[0]):
                if action not in RRTS.get_all_actions():
482                 RRTS.add_internal_action(action)
                next = self.get_next_state(todo[0], action)
484             RRTS.add_transition(todo[0], action, next)
                if next not in todo and next not in done:
486                 todo.append(next)
            done.add(todo[0])
488         todo.pop(0)
        return RRTS
490
    def is_automaton_reduced(self):
492     """Tells you if an automaton is already reduced.

494     This operation is as expensive as generating a reduced automaton and
        should be avoided.
496     """
        if (len(self.states) == len(self.get_reachable_states()) and
498         len(self.transitions) == len(self.get_useful_transitions())):
                return True
500     else:
                return False
502
    def is_automaton_compatible(self):
504     """Returns true if compatible according to definition.

506     This requires the following:
            - No livelocks
508         - No message loss (always receptive)
            - Deadlock-free
```

```
510
        Note: this assumes the automaton was constructed with a synchronous
512     production function that records incompatibilities, or that it is an
        elementary automaton which has no "history" for incompatibilities.
514     """
        if self.detect_livelock(quiet=True):
516         return False
        if (len(self.message_losses) + len(self.deadlocks)) == 0:
518         return True
        return False
```

../source_code/TA.py

```
#! /usr/bin/env python

from TA import TA
import itertools


def take_transition(current, action, participants, components, result):
    """Follow a transition (current, action)->next

    Computes the next-state arrived at by executing 'action', then adds
    the transition to result.transitions, and if necessary also adds the 'next'
    state to result.states.

    The 'participants' are the components of 'result' that take part in the
    transition.

    Returns the 'next' state to the caller.
    """
    next = tuple(current[s] if s not in participants else
        components[s].get_next_state(current[s], action)
        for s in range(len(components)))
    result.add_transition("_".join(current), action, "_".join(next))
    return next

def add_if_needed(state, todo, done):
    """Adds the state to 'todo' if it's not been encountered before.
    """
    if state not in todo and state not in done:
        todo.append(state)

def synchronous_product_effort(components = [], find_all_contradictions=True):
    """Compute the synchronous product of a list of components.

    Will also determine message loss and deadlock situations.

    Does not pay attention to livelocks. Livelocks will not cause the algorithm
    trouble. Note that livelocks could have been checked and
    prevented at the component preprocessing stage. If the components are
    livelock-free, then the synchronous product will also be livelock-free.
    (If the components were not livelock-free, then the synchronous product
    might still be livelock-free, because it is a Reduced system.)

    If the result is livelock-free, and len(result.message_losses) ==
    len(result.deadlocks) == 0, then results.states is a compatibility relation.
    """
    effort = 0
    # check sanity of input
    if len(components) == 0:
        print "ERROR: no components."
        return None
    elif len(components) == 1:
        return components[0]
    # check for composability
    for pair in list(itertools.combinations(components, 2)):
        if pair[0] == pair[1]:
            print "Error: synchronizing duplicate component"
            return None
        if not pair[0].internal_actions.isdisjoint(pair[1].get_all_actions()):
            print "ERROR: shared internal action"
            return None
        if not pair[1].internal_actions.isdisjoint(pair[0].get_all_actions()):
            print "ERROR: shared internal action"
            return None
    result = TA()
    result.name = "XR_" + "_".join([c.name for c in components])
    # fill alphabets
    # internal actions will not be added to 'result' unless they prove useful.
    internal_actions = set().\
        union(*[c.internal_actions for c in components])
    # external actions will always be added to 'result'.
    result.output_actions = set().\
        union(*[c.output_actions for c in components])
    result.input_actions = set().\
```

```
        union(*[c.input_actions for c in components]).\
75      difference(result.output_actions)
    # compute implied sub-alphabets and domains
77  external_actions = result.output_actions.union(result.input_actions)
    communicating_actions = set()
79  shared_input_actions = set() # shared input, but not output
    shared_output_actions = set() # shared output, but not input
81  noncommunicating_input_actions = set()
    noncommunicating_output_actions = set()
83  inp_dom = {}
    out_dom = {}
85  int_dom = {}
    for action in result.get_all_actions():
87      if action in internal_actions:
            continue # useful internal actions will be added later
89      inp = {c for c in range(len(components))
                if action in components[c].input_actions}
91      out = {c for c in range(len(components))
                if action in components[c].output_actions}
93      Li = len(inp)
        Lo = len(out)
95      if Li > 0:
            inp_dom[action] = inp
97      if Lo > 0:
            out_dom[action] = out
99      if Li * Lo > 0:
            communicating_actions.add(action)
101     elif Li == 1:
            noncommunicating_input_actions.add(action)
103     elif Li > 1:
            shared_input_actions.add(action)
105     elif Lo == 1:
            noncommunicating_output_actions.add(action)
107     elif Lo > 1:
            shared_output_actions.add(action)
109 for c in range(len(components)):
        for action in components[c].internal_actions:
111         int_dom[action] = c
    # initialize statespace
113 done = set()
    todo = [tuple(c.initial_state for c in components)]
115 result.initial_state = "_".join(todo[0])
    result.add_state(result.initial_state)
117 # explore the statespace
    while len(todo) > 0:
119     # explore one particular state
        possible_deadlock = "unknown"
121     current = todo.pop()
        done.add(current)
123     current_state = "_".join(current)
        for action in communicating_actions:
125         effort += len(out_dom[action])
            effort += len(inp_dom[action])
127         if all(components[c].is_action_enabled(current[c], action)
                for c in out_dom[action]): # output domain enabled
129             if all(components[c].is_action_enabled(current[c], action)
                    for c in inp_dom[action]): # input domain enabled
131                 next = take_transition(current, action,
                        set().union(inp_dom[action],
133                     out_dom[action]), components, result)
                    add_if_needed(next, todo, done)
135                 possible_deadlock = "averted"
                else: # message loss
137                 result.message_losses.add((current_state, action))
                    if find_all_contradictions == False:
139                     return (result, effort)
            else: # output not domain enabled
141             if all(components[c].is_action_enabled(current[c], action)
                    for c in inp_dom[action]): # input domain enabled
143                 if possible_deadlock == "unknown":
                        possible_deadlock = "threat"
145     for action in shared_input_actions:
            effort += len(inp_dom[action])
```

```
147             if all(components[c].is_action_enabled(current[c], action)
                    for c in inp_dom[action]): # input domain enabled
149                 next = take_transition(current, action,
                        inp_dom[action], components, result)
151                 add_if_needed(next, todo, done)
                    possible_deadlock = "averted"
153         for action in shared_output_actions:
                effort += len(out_dom[action])
155             if all(components[c].is_action_enabled(current[c], action)
                    for c in out_dom[action]): # output domain enabled
157                 next = take_transition(current, action,
                        out_dom[action], components, result)
159                 add_if_needed(next, todo, done)
                    possible_deadlock = "averted"
161         for action in noncommunicating_input_actions:
                effort += len(inp_dom[action])
163             if all(components[c].is_action_enabled(current[c], action)
                    for c in inp_dom[action]): # input domain enabled
165                 next = take_transition(current, action,
                        inp_dom[action], components, result)
167                 add_if_needed(next, todo, done)
                    possible_deadlock = "averted"
169         for action in noncommunicating_output_actions:
                effort += len(out_dom[action])
171             if all(components[c].is_action_enabled(current[c], action)
                    for c in out_dom[action]): # output domain enabled
173                 next = take_transition(current, action,
                        out_dom[action], components, result)
175                 add_if_needed(next, todo, done)
                    possible_deadlock = "averted"
177         for action in internal_actions:
                effort += 1
179             if components[int_dom[action]].is_action_enabled(
                    current[int_dom[action]], action):
181                 result.internal_actions.add(action) # useful => include
                    next = take_transition(current, action,
183                     {int_dom[action]}, components, result)
                    add_if_needed(next, todo, done)
185                 possible_deadlock = "averted"
            if possible_deadlock == "threat": # threat was not averted
187             result.deadlocks.add(current_state)
                if find_all_contradictions == False:
189                 return (result, effort)
    # post-exploration checks
191 if result.sanity_check():
        return (result, effort)
193 else:
        print "Error: resulting TA is not sane."
195     return None
```

../source_code/synchronize.py

```python
# module to make graphs for Team Automata
# strongly inspired by http://matthiaseisen.com/articles/graphviz/

# external modules
import functools
import graphviz
import os

# local modules
import TA

# default format
my_format = 'svg'

# shorthand
digraph = functools.partial(graphviz.Digraph, format=my_format)

def changeformat(my_format):
    """ Change the output format
    """
    return functools.partial(graphviz.Digraph, format=my_format)

def add_nodes(graph, nodes):
    """ Add nodes to the graph

    Nodes are either entered as a string, or as a tuple.

    When entered as a tuple, it is done in the form (string, {}) and the second
    element holds style information for graphviz.
    """
    for n in nodes:
        if isinstance(n, tuple):
            graph.node(n[0], **n[1])
        else:
            graph.node(n)
    return graph

def add_edges(graph, edges):
    """ Adds edges between nodes.

    Edges are either entered as a string, or as a tuple.

    When entered as a tuple, it is done in the form (string, {}) and the second
    element holds style information for graphviz.
    """
    for e in edges:
        if isinstance(e[0], tuple):
            graph.edge(*e[0], **e[1])
        else:
            graph.edge(*e)
    return graph


def make_digraph(name, nodes, edges):
    """ Make a directed graph from a set of nodes and edges connecting them.
    """
    add_edges(
        add_nodes(digraph(), nodes),
        edges
    ).render(name)


def makeGraph(subject = TA.TA()):
    """ Make the graph for a  TA.

    The argument should be a TA object.
    """
    nodes = list(subject.states.difference(set(subject.initial_state)))
    nodes.append((subject.initial_state, {'penwidth':'3'}))
    edges = []
    for t in subject.transitions:
        if t[1] in subject.internal_actions:
            edges.append(((t[0], subject.transitions[t]), {'label':'  '+t[1]+'@  '}))
```

```
        elif t[1] in subject.input_actions:
            edges.append(((t[0], subject.transitions[t]), {'label':'  '+t[1]+'?  '}))
        elif t[1] in subject.output_actions:
            edges.append(((t[0], subject.transitions[t]), {'label':'  '+t[1]+'!  '}))
        else:
            print "Error: transition using action of unknown type:", t, subject.transitions[t]
    make_digraph(subject.name, nodes, edges)
    os.remove(subject.name)
```

../source_code/graph.py

```
import sys
import graph
import TA

# this script is intended to allow us to make graphs from the commandline.
# usage:
# python graph_commandline.py filename
# produces a graph with the TA's actual name as 'name.svg', regardless of the filename

if len(sys.argv) == 3:
    graph.digraph = graph.changeformat(sys.argv[1])
    subject = TA.TA(sys.argv[2])
else:
    subject = TA.TA(sys.argv[1])
graph.makeGraph(subject)
```

../source_code/graph_commandline.py

```
#! /usr/bin/env python

import itertools
import random
import operator
import math
from pyparsing import nestedExpr

from TA import TA
import synchronize




class Scheduler:

    def __init__(self, filename=None, strategy=None, find_all_contradictions=True):
        """ Create a new schedule

        If a filename is given, that schedule will be loaded. Otherwise an empty
        schedule is returned, which can be used to resolve sub-schedules.
        """
        self.components = {}
        self.combinations = {}
        self.ordered_components = []
        self.initial_effort = 0
        self.set_strategy(strategy)
        self.contradictions = find_all_contradictions
        if filename != None:
            goal = self.read_schedule_file(filename)
            self.initial_effort = self.resolve_components_effort(goal)
        else:
            return

    def read_schedule_file(self, filename):
        """ Read in a schedule file and parse it
        """
        data = ""
        with open (filename, "r") as myfile:
            goal=myfile.read().replace('\n', '').replace(',', ' ')\
                .replace('  ', ' ')
        schedule = nestedExpr('(',')').parseString(goal).asList()[0]
        return schedule

    def resolve_components_effort(self, goal):
        """ Distinguish base components and subschedules in the schedule.

        Base components are loaded. Subschedules are recursively resolved into
        components, which are then added.

        The effort expended on subschedules is inherited by the superschedule.
        """
        total_effort = 0
        for item in range(len(goal)):
            if isinstance(goal[item], str):
                self.load_component(goal[item])
            if isinstance(goal[item], list):
                subteam = Scheduler(None, self.strategy)
                effort = subteam.resolve_components_effort(goal[item])
                total_effort += effort
                result, effort = subteam.run_effort()
                total_effort += effort
                result.save()
                self.load_component(result.name)
        return total_effort

    def load_component(self, name):
        """ Load a component from file and preprocess it.

        Components are reduced and then checked for livelocks. Even if a livelock
        is found, the loading continues.
```

```
        The detect_livelock method of TA will report what livelock was found;
        this function reports in which component it was found.
        """
        temp = TA(name + ".out")
        temp = temp.get_reduced_version()
        if temp.detect_livelock():
            print "    in component:", name
            if self.contradictions == False:
                print "Stopping now, because complete search (despite finding a contradiction
    already) is OFF."
                exit(0)
        self.components["R_"+name] = temp
        self.ordered_components.append("R_"+name)

    def collaborates_nicely(self, choice):
        """ Tests if a choice of next job enjoys collaboration of remainder.

        Will return True if the automata of choice share no output action with
        the remaining components that is not also an input action in some of
        the remaining components; else False.
        """
        left_out = set().union(self.components[choice[0]].output_actions,
            self.components[choice[1]].output_actions)
        right_out = set()
        for c in self.components:
            if c == choice[0] or c == choice[1]:
                continue
            right_out = right_out.union(self.components[c].output_actions)
        if len(left_out.intersection(right_out)) > 0:
            return False
        return True

    def get_next_job(self, previous, choice):
        """ Determine what components to try to synchronize next.

        'previous' is the previous synchronization result, which is examined
        here.

        If 'previous' is compatible:
            - The components used to make 'previous' are removed from the
            remaining unsynchronized components
            - All combinations using the old components are removed from the
            list of possibilities
            - 'previous' is added to the set of remaining unsynchronized
            components
            - Combinations with the new component are computed and added to the
            list of possibilities
        If 'previous' is NOT compatible:
            - It is not added to the set of remaining unsynchronized components
            - It's constituent components stay in the set of remaining
            unsynchronized components
            - The combination of its constituent components is removed from
            the list of possible combinations

        If viable combinations remain they're sorted by fitness. One by one they
        will be tested to see if they are output-action-complementary. The first
        one meeting this criterium is returned.
        Else, a last-ditch attempt is made to synchronize all remaining
        components at once.
        If that also fails, there is no next job.
        """
        # STEP 1: process previous job
        if isinstance(previous, TA):
            # if previous pairing turned out incompatible
            if previous.is_automaton_compatible():
                # if previous pairing came from combination
                if (choice[0].name, choice[1].name) in self.combinations:
                    # dispose of the incompatible combination
                    del self.combinations[(choice[0].name, choice[1].name)]
                # else a final attempt turned out incompatible
                else:
                    return None # no more jobs to try
```

```
            else:
                # if previous pairing came from combination
                if (choice[0].name, choice[1].name) in self.combinations:
                    # flush obsolete combinations and synchronized components
                    obsolete = [x for x in self.combinations.keys() if (
                            x[0] == choice[0].name or
                            x[1] == choice[0].name or
                            x[0] == choice[1].name or
                            x[1] == choice[1].name)]
                    for item in obsolete:
                        del self.combinations[item]
                    del self.components[choice[0].name]
                    del self.components[choice[1].name]
                    # add and analyze the new component
                    self.components[previous.name] = previous
                    self.ordered_components.append(previous.name)
                    for component in self.components:
                        if component != previous.name:
                            self.analyze(component, previous.name)
                # success was the result of final synchronization
                else:
                    return None # no more jobs needed

        # STEP 2: select next job
        # if viable pairs are still available
        if len(self.combinations) > 0:
            # get choices sorted by fitness
            choices = sorted(self.combinations, key=self.combinations.get,
                             reverse=True)
            # now return the highest-fitness possible choice that collaborates nicely
            for choice in choices:
                if self.collaborates_nicely(choice):
                    return [self.components[choice[0]], self.components[choice[1]]]
            # if no pair was found that collaborates nicely, try all at once
            self.combinations.clear()
            return list(self.components.values())
        # elif 3+ components remain but no combinations, try all at once
        elif len(self.components) > 2:
            return list(self.components.values())
        # else give up
        else:
            return None

    def run_effort(self):
        """ Synchronize the schedule.

        This function assumes that all subschedules have already been resolved.
        It iterates by asking get_next_job for a new job, and submitting the
        result of the previous job, until no more jobs are forthcoming. Then
        what remains is the synchronous product of the schedule including
        information about its compatibility.

        It sends jobs to the 'synchronize.synchronous_product_effort' function
        which returns the result of synchronization and the effort expended.

        Effort expended on jobs that didn't succeed is still counted.
        """
        total_effort = self.initial_effort
        for pair in list(itertools.combinations(self.components, 2)):
            self.analyze(pair[0], pair[1])
        result = None
        choice = None
        while True:
            choice = self.get_next_job(result, choice)
            if choice == None:
                break
            result, effort = synchronize.synchronous_product_effort(choice, self.
    contradictions)
            total_effort += effort
        return (result, total_effort)

    def set_strategy(self, strategy):
        """ Set a strategy for this schedule.
```

```
        It associates a strategy function to the analyze(A, B) function, so that
        any calls to analyze use the associated function.
        """
        self.strategy = strategy # record for reuse by recursion over subteams
        if strategy == "action sim":
            self.analyze = self.strategy_action_similarity
        elif strategy == "action dissim":
            self.analyze = self.strategy_action_dissimilarity
        elif strategy == "fifo":
            self.analyze = self.strategy_fifo
        elif strategy == "lifo":
            self.analyze = self.strategy_lifo
        elif strategy == "random":
            self.analyze = self.strategy_random
        elif strategy == "Q min":
            self.analyze = self.strategy_Q_min
        elif strategy == "Q max":
            self.analyze = self.strategy_Q_max
        elif strategy == "all at once":
            self.analyze = self.strategy_all_at_once
        else:
            print "ERROR: unknown strategy."
            exit(1)


    def strategy_random(self, A1, A2):
        """ Combinations receive a random fitness value.
        """
        self.combinations[(A1, A2)] = random.random()

    def strategy_action_similarity(self, A1, A2):
        """ Pairs receive higher scores if they have more actions in common.
        """
        AA1 = self.components[A1]
        AA2 = self.components[A2]
        total = len(set().union(AA1.get_all_actions(), AA2.get_all_actions()))
        shared = len(AA1.get_all_actions().intersection(AA2.get_all_actions()))
        self.combinations[(A1, A2)] = shared/float(total)

    def strategy_action_dissimilarity(self, A1, A2):
        """ Pairs receive higher scores if they have less actions in common.
        """
        AA1 = self.components[A1]
        AA2 = self.components[A2]
        total = len(set().union(AA1.get_all_actions(), AA2.get_all_actions()))
        shared = len(AA1.get_all_actions().intersection(AA2.get_all_actions()))
        self.combinations[(A1, A2)] = -(shared/float(total))

    def strategy_fifo(self, A1, A2):
        """ First In First Out synchronization of pairs.

        Results of synchronization end up at the back of the queue, so first
        all base components are used, then all first-generation product
        components and so forth.
        """
        B1 = None
        B2 = None
        obsolete = set()
        if len(self.combinations) < 1:
            # select the leftmost component that still exists
            for item in range(len(self.ordered_components)):
                if self.ordered_components[item] in self.components.keys():
                    B1 = self.ordered_components[item]
                    break
                else:
                    obsolete.add(item)
            # select the second to leftmost component that still exists
            for item in range(len(self.ordered_components)):
                if self.ordered_components[item] in self.components.keys():
                    if self.ordered_components[item] != B1:
                        B2 = self.ordered_components[item]
                        break
```

```
                else:
                    obsolete.add(item)
            # get rid of leftmost components that no longer exist
            for item in sorted(obsolete, reverse=True):
                del self.ordered_components[item]
            self.combinations[(B1, B2)] = 1

    def strategy_lifo(self, A1, A2):
        """ Last In First Out synchronization of pairs.

        Effectively, the result of the latest synchronization is immediately
        used to synchronize again, with the rightmost base component.
        """
        B1 = None
        B2 = None
        obsolete = set()
        if len(self.combinations) < 1:
            # select rightmost component that still exists
            for item in reversed(range(len(self.ordered_components))):
                if self.ordered_components[item] in self.components.keys():
                    B1 = self.ordered_components[item]
                    break
                else:
                    obsolete.add(item)
            # select second to rightmost component that still exists
            for item in reversed(range(len(self.ordered_components))):
                if self.ordered_components[item] in self.components.keys():
                    if self.ordered_components[item] != B1:
                        B2 = self.ordered_components[item]
                        break
                else:
                    obsolete.add(item)
            # remove rightmost components that no longer exist
            for item in sorted(obsolete, reverse=True):
                del self.ordered_components[item]
            self.combinations[(B1, B2)] = 1

    def strategy_Q_min(self, A1, A2):
        """ Take the pair of components with the smallest combined state space
        """
        self.combinations[(A1, A2)] = -(len(self.components[A1].states) * \
            len(self.components[A2].states))

    def strategy_Q_max(self, A1, A2):
        """ Take the pair of components with the largest combined state space
        """
        self.combinations[(A1, A2)] = len(self.components[A1].states) * \
            len(self.components[A2].states)

    def strategy_all_at_once(self, A1, A2):
        """ Do not make pairs at all.

        As a result, all components will be synchronized as if all viable pairs
        had already been eliminated, i.e. immediately try too synchronize all
        components together at once.
        """
        if len(self.components) == 2:
            self.combinations[(list(self.components)[0],
                               list(self.components)[1])] = 1
```

../source_code/scheduler.py

```
import sys
import scheduler

schedule = None
strategy = "fifo"
complete = False
for elem in sys.argv[1:]:
    if elem.startswith("strategy="):
        strategy = elem.replace("strategy=", "")
    if elem.startswith("complete="):
        if elem.replace("complete=", "") == "True" or elem.replace("complete=", "") == "true":
            complete = True
    if elem.startswith("schedule="):
        schedule = elem.replace(".scd", "").replace("schedule=", "")

print "SCHEDULE: ", schedule
print "STRATEGY: ", strategy
print "COMPLETE: ", complete

if schedule == None:
    print "ERROR: schedule parameter required. Usage: \"schedule=[schedulename]\""
    exit(1)

worker = scheduler.Scheduler(schedule+".scd", strategy=strategy, find_all_contradictions=
    complete)
result, effort = worker.run_effort()
result.save('X'+schedule)

#result.screendump()

print "Effort: ", effort
if result.is_automaton_compatible():
    print "This is a compatible system."
else:
    print "This is NOT a compatible system."
```

../source_code/schedule_commandline.py

```
import scheduler
import graph

schedules = ['workplace', 'coffeeTime', 'tokenRing','tokenRing2','settlement',
    'wedding','passing']
strategies = ['action sim', 'action dissim', 'fifo', 'lifo',
    'Q min', 'Q max', 'all at once']

stats = []
find_all_contradictions=False

for schedule in schedules:
    row = [schedule]
    print "\n",schedule.upper()
    for strategy in strategies:
        worker = scheduler.Scheduler(schedule+".scd", strategy=strategy,
    find_all_contradictions=find_all_contradictions)
        result, effort = worker.run_effort(find_all_contradictions=find_all_contradictions)
        result.save('X'+schedule)
        if strategy == strategies[0]:
            row.append(result.is_automaton_compatible())
#             graph.makeGraph(result)
#             result.screendump()
        row.append(effort)
    stats.append(row)

offset = 15

print "".rjust(offset), "Compatible?".rjust(offset),
for strategy in strategies: print strategy.rjust(offset),
print
for row in stats:
    for cell in row:
        print str(cell).rjust(offset),
    print

# output as tex table
with open('experiment.tex', 'w') as f:
    f.write("\\begin{myTable}{| X[1.5] | X[1.5] | ")
    for x in range(len(stats[0])-2): f.write("X[r] | ")
    f.write("}\n\\hline\n")
    f.write(" &".rjust(offset))
    f.write("\\footnotesize Compatible? &".rjust(offset))
    for s in range(len(strategies)-1):
        f.write(("\\footnotesize "+strategies[s]+" &").rjust(offset))
    f.write(("\\footnotesize "+strategies[-1]).rjust(offset))
    f.write("\\\\\n\\hline\n")
    for row in stats:
        for cell in range(len(row)-1):
            f.write(("\\footnotesize "+str(row[cell])+" &").rjust(offset))
        f.write("\\footnotesize "+str(row[-1]).rjust(offset))
        f.write("\\\\\n\\hline\n")
    f.write("\\end{myTable}")

# output as CSV
with open('experiment.csv', 'w') as f:
    f.write(", Compatible?,")
    for s in strategies:
        f.write(s + ",")
    f.write("\n")
    for row in stats:
        for cell in row:
            f.write(str(cell) + ",")
        f.write("\n")
```

../source_code/experiment.py

B Automata test sets

B.1 Boss/Employee

```
.model boss
.inputs work_harder
.outputs get_raise
.internal
.state graph
    unhappy work_harder happy
    happy get_raise happy
.marking {unhappy}
.end
```

../source_code/cases/boss.out

```
.model employee
.inputs get_raise
.outputs work_harder
.internal
.state graph
    unhappy get_raise happy
    happy work_harder happy
.marking {unhappy}
.end
```

../source_code/cases/employee.out

B.2 Coffee

```
.model drinker
.inputs serve_coffee
.outputs press_button
.internal drink_coffee
.state graph
    thirsty press_button waiting
    waiting serve_coffee happy
    happy drink_coffee thirsty
.marking {thirsty}
.end
```

../source_code/cases/drinker.out

```
.model machine
.inputs press_button
.outputs serve_coffee
.internal clean_filters
.state graph
    ready press_button brewing
    brewing serve_coffee dirty
    dirty clean_filters ready
.marking {ready}
.end
```

../source_code/cases/machine.out

B.3 Settlement

```
.model investor1
.inputs placementConfirmation cancellationConfirmation
.outputs placementOrder1 cancellationOrder1
.internal
.state graph
    start1 placementOrder1 waiting1
    waiting1 cancellationOrder1 waiting1Cancellation
    waiting1Cancellation cancellationConfirmation finished1
    waiting1 placementConfirmation finished1
    start1 cancellationConfirmation finished1
    waiting1 cancellationConfirmation finished1
.marking {start1}
.end
```

../source_code/cases/investor1.out

```
.model investor2
.inputs placementConfirmation cancellationConfirmation
.outputs placementOrder2 cancellationOrder2
.internal acceptPlacementConfirmation
.state graph
    start2 placementOrder2 waiting2
    waiting2 cancellationOrder2 waiting2Cancellation
    waiting2Cancellation cancellationConfirmation finished2
    waiting2 placementConfirmation wrong
    wrong cancellationOrder2 waiting2Cancellation
    start2 cancellationConfirmation finished2
    waiting2 cancellationConfirmation finished2
    wrong acceptPlacementConfirmation finished2
.marking {start2}
.end
```

../source_code/cases/investor2.out

```
.model custodian
.inputs placementOrder1 placementOrder2 cancellationOrder1 cancellationOrder2
.outputs placementConfirmation cancellationConfirmation
.internal
.state graph
    No_No placementOrder1 Yes_No
    No_No placementOrder2 No_Yes
    No_Yes placementOrder1 Yes_Yes
    Yes_No placementOrder2 Yes_Yes
    Yes_Yes placementConfirmation finished
    Yes_Yes cancellationOrder1 cancelling
    Yes_Yes cancellationOrder2 cancelling
    Yes_No cancellationOrder1 cancelling
    No_Yes cancellationOrder2 cancelling
    cancelling cancellationOrder1 cancelling
    cancelling cancellationOrder2 cancelling
    cancelling cancellationConfirmation finished
    cancelling placementOrder1 cancelling
    cancelling placementOrder2 cancelling
.marking {No_No}
.end
```

../source_code/cases/custodian.out

B.4 Token Ring

```
.model user1
.inputs grant1 tick
.outputs request1
.internal enter1 go1
.state graph
    offline enter1 online
    offline tick offline
    online tick online
    waiting tick online
    online request1 waiting
    waiting grant1 online
    online go1 offline
.marking {online}
.end
```

../source_code/cases/user1.out

```
.model station1a
.inputs token1 tick request1
.outputs grant1
.internal
.state graph
    waiting tick waiting
    waiting token1 haveToken
    haveToken request1 willGrant
    haveToken tick waiting
    willGrant grant1 haveGranted
    haveGranted tick waiting
.marking {waiting}
.end
```

../source_code/cases/station1a.out

```
.model station1b
.inputs tick token1
.outputs token2
.internal
.state graph
    waiting tick waiting
    waiting token1 haveToken
    haveToken tick mustPass
    mustPass token2 waiting
.marking {waiting}
.end
```

../source_code/cases/station1b.out

```
.model user2
.inputs grant2 tick
.outputs request2
.internal enter2 go2
.state graph
    offline enter2 online
    offline tick offline
    online tick online
    waiting tick online
    online request2 waiting
    waiting grant2 online
    online go2 offline
.marking {online}
.end
```

../source_code/cases/user2.out

```
.model station2a
.inputs token2 tick request2
.outputs grant2
.internal
.state graph
```

```
    waiting tick waiting
    waiting token2 haveToken
    haveToken request2 willGrant
    haveToken tick waiting
    willGrant grant2 haveGranted
    haveGranted tick waiting
.marking {waiting}
.end
```

../source_code/cases/station2a.out

```
.model station2b
.inputs tick token2
.outputs token3
.internal
.state graph
    waiting tick waiting
    waiting token2 haveToken
    haveToken tick mustPass
    mustPass token3 waiting
.marking {waiting}
.end
```

../source_code/cases/station2b.out

```
.model user3
.inputs grant3 tick
.outputs request3
.internal enter3 go3
.state graph
    offline enter3 online
    offline tick offline
    online tick online
    waiting tick online
    online request3 waiting
    waiting grant3 online
    online go3 offline
.marking {online}
.end
```

../source_code/cases/user3.out

```
.model station3a
.inputs token3 tick request3
.outputs grant3
.internal
.state graph
    waiting tick waiting
    waiting token3 haveToken
    haveToken request3 willGrant
    haveToken tick waiting
    willGrant grant3 haveGranted
    haveGranted tick waiting
.marking {haveToken}
.end
```

../source_code/cases/station3a.out

```
.model station3b
.inputs tick token3
.outputs token1
.internal
.state graph
    waiting tick waiting
    waiting token3 haveToken
    haveToken tick mustPass
    mustPass token1 waiting
.marking {haveToken}
.end
```

../source_code/cases/station3b.out

B.5 Wedding

```
.model bride
.inputs do_you_groom groom_I_do groom_I_dont do_you_bride poll_objections objection
    pronounce_marriage
.outputs bride_enters bride_I_do bride_I_dont kiss
.internal
.state graph
    outside bride_enters bw1
    bw1 do_you_groom bw2
    bw2 groom_I_do bw3
    bw2 groom_I_dont problem
    bw3 do_you_bride ms1
    ms1 bride_I_do bw4
    ms1 bride_I_dont problem
    bw4 poll_objections bw5
    bw5 objection problem
    bw5 pronounce_marriage ms2
    ms2 kiss finished
.marking {outside}
.end
```

../source_code/cases/bride.out

```
.model groom
.inputs bride_enters do_you_groom do_you_bride bride_I_do bride_I_dont poll_objections
    objection pronounce_marriage
.outputs groom_I_do groom_I_dont kiss
.internal hesitate
.state graph
    gw1 bride_enters gw2
    gw2 do_you_groom ms1
    ms1 hesitate ms1
    ms1 groom_I_do gw3
    ms1 groom_I_dont problem
    gw3 do_you_bride gw4
    gw4 bride_I_do gw5
    gw4 bride_I_dont problem
    gw5 poll_objections gw6
    gw6 objection problem
    gw6 pronounce_marriage ms2
    ms2 kiss finished
.marking {gw1}
.end
```

../source_code/cases/groom.out

```
.model minister
.inputs bride_enters groom_I_do groom_I_dont bride_I_do bride_I_dont objection
.outputs do_you_groom do_you_bride poll_objections pronounce_marriage
.internal
.state graph
    mw1 bride_enters ms1
    ms1 do_you_groom mw2
    mw2 groom_I_do ms2
    mw2 groom_I_dont problem
    ms2 do_you_bride mw3
    mw3 bride_I_do ms3
    mw3 bride_I_dont problem
    ms3 poll_objections mw4
    mw4 objection problem
    mw4 pronounce_marriage finished
.marking {mw1}
.end
```

../source_code/cases/minister.out

B.6 Passing the Ball

```
.model A1
.inputs a5
.outputs a1
.internal
.state graph
    ball a1 noball
    noball a5 ball
.marking {ball}
.end
```

../source_code/cases/A1.out

```
.model A2
.inputs a1
.outputs a2
.internal
.state graph
    noball a1 ball
    ball a2 noball
.marking {noball}
.end
```

../source_code/cases/A2.out

```
.model A3
.inputs a2
.outputs a3
.internal
.state graph
    noball a2 ball
    ball a3 noball
.marking {noball}
.end
```

../source_code/cases/A3.out

```
.model A4
.inputs a3
.outputs a4
.internal
.state graph
    noball a3 ball
    ball a4 noball
.marking {noball}
.end
```

../source_code/cases/A4.out

```
.model A5
.inputs a4
.outputs a5
.internal
.state graph
    noball a4 ball
    ball a5 noball
.marking {noball}
.end
```

../source_code/cases/A5.out