Regulated Reactive Robotics: A Formal Framework

Jered Vroon[a,*], Iris van Rooij[a,b], Todd Wareham[c], Pim Haselager[a,b]

[a] *Department of Artificial Intelligence, Radboud University, Montessorilaan 3, 6525 HR, Nijmegen, The Netherlands*
[b] *Donders Institute for Brain, Cognition, and Behaviour, Radboud University, Kapittelweg 29, 6525 EN Nijmegen, The Netherlands*
[c] *Department of Computer Science, Memorial University of Newfoundland, A1B 3X5, St.John's, Newfoundland, Canada*

Abstract

In this paper we introduce the Regulated Reactive approach to robotics, which adds regulation (local control) to the existing Reactive Approach. Furthermore, we present a formal framework for structure-level descriptions of systems and relate those to hardware requirements. We prove that for any given behavior there exists a regulated reactive system that requires at most as many resources as an optimal reactive or hierarchical system. Furthermore, we show that for particular behaviors in which common conditions are true, reactive and hierarchical systems require more space resources than necessary as for those tasks there exist regulated reactive systems that require less resources. This makes regulated reactive systems an attractive framework for robotics design.

Keywords: Reactive Robotics, Hierarchical Robotics, Regulated Reactive Robotics

Many approaches have been suggested and used to design fast and accurate robot software. This paper introduces a new such approach that can help bridge the divide between what traditionally are two of the main approaches, Reactive Robotics and Hierarchical Robotics.

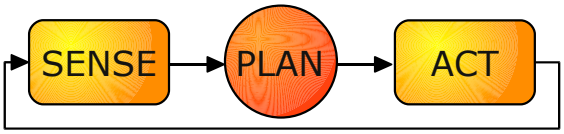
Many of the first attempts in robotics can be classified as Hierarchical Robotics (e.g. Shakey [1]), which developed alongside a similar approach to cognition (e.g. [2]). They commonly have a modular design with preprocessing modules, a central planner, and postprocessing modules (Figure 1a). The preprocessing units derive higher order information (e.g. the presence and position of an obstacle) from the sensory data. Then, the central planner uses all that information to come up with a planning, most commonly by using some sort of logical reasoning. Finally, this planning is turned into actual motor commands by the postprocessing units, after which the cycle can start over.

Reasonable as this approach may be, it often turned out to be too slow to handle everyday tasks such as walking effectively - even though it is quite effective when much reasoning is required. That is, the continuous cycle of preprocessing, (usually tedious and time-consuming) planning and postprocessing commonly is not flexible and fast enough to make all the quick minor adaptations that are required for such everyday tasks.

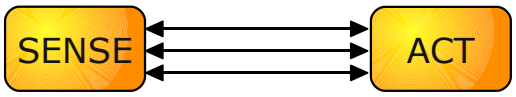
In response to this challenge to deal efficiently with everyday tasks, Reactive Robotics was introduced [3]. Reactive Robotics does away with the central planner alltogether. Instead it uses several behavioral layers that provide basic couplings between parts of the input and parts of the output (e.g. one such layer could lead a robot to approach observed food) (Figure 1b). When combined, the outputs these behavioral layers provide, result in intricate behaviors. Even though it is obviously less apt at complex reasoning, reactive systems have been succesfull in producing everyday behaviors such as basic navigation and obstacle avoidance.

Since many different variations of both Reactive and Hierarchical Robotics exist, we will not claim that the above descriptions capture all those variations. However, we do feel that these descriptions capture the

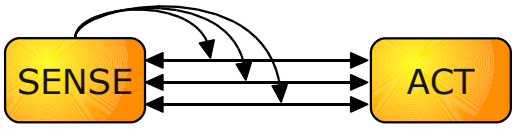
*Corresponding author
Email addresses: jeredVroon@student.ru.nl (Jered Vroon), I.vanRooij@donders.ru.nl (Iris van Rooij), harold@cs.mun.ca (Todd Wareham), W.Haselager@donders.ru.nl (Pim Haselager)



(a) hierarchical



(b) reactive



(c) regulated reactive

Figure 1: In these figures the different robotic systems here discussed are displayed. In an hierarchical system a central planner (without different layers) takes prepocessed input and returns to be postprocessed output. In a reactive system, multiple distinct, independent layers all produce (overlapping) parts of the output from parts of the input (no planning). In a regulated reactive system, we have a reactive system with on top of that input-driven regulation (so no central planning).

essential features and their associated weaknesses. Therefore, we will stick to these descriptions of 'basic' Reactive and Hierarchical Robotics.

The remainder of this paper is organized as follows. First, Regulated Reactive Robotics will be introduced (Section 1). Then, a formal framework for giving structure level descriptions of systems will be introduced (Section 2 and 3) and related to the hardware requirements of those systems (Section 4). It is then shown that a Brute Force approach has unfeasible hardware requirements (Section 5) and that these hardware requirements can be reduced by generalizing (Section 6). Various ways to generalize are then introduced (Section 7) and related to Reactive, Hierarchical and Regulated Reactive Robotics (Section 8). Regulated Reactive Robotics is found to be more general and more resource efficient than those two existing approaches (Section 9).

1. Regulated Reactive Robotics

Embodied Embedded Cognition (EEC) is a relatively young approach in cognitive science. In contrast to the more traditional approaches, it stresses the importance of the body and environment of an agent to the behavior it displays. That is, it emphasizes that patterns in the environment can often be exploited by an agent to use simple, basic, automatic behaviors instead of deliberative, rational planning (i.e. "to be lazy") [4].

This resembles the Reactive approach quite a bit, but adds to that simple, basic, automatic, lazy regulation. Though we will formalize this in more detail later on, one can think of this as local influencing of behavioral layers, rather than central control (such as in the Hierarchical approach). This is much like the way in which traffic lights regulate the traffic in their proximity, as opposed to more central control (e.g. air traffic control). Recently van Dijk et al. [5] suggested such regulation as a means to extend the Reactive approach beyond automatic behaviors while trying to avoid the problems with central reasoning. Simulations with such regulation have shown it more [6] or less [7] succesfull.

In this paper, we flesh out this proposal to what we call Regulated Reactive Robotics. Though we will give a more formal definition later on, Regulated Reactive Robotics in essence comprises two elements (see Figure 1c). The first element is a set of behavioral layers, like those in Reactive Robotics, that provide all

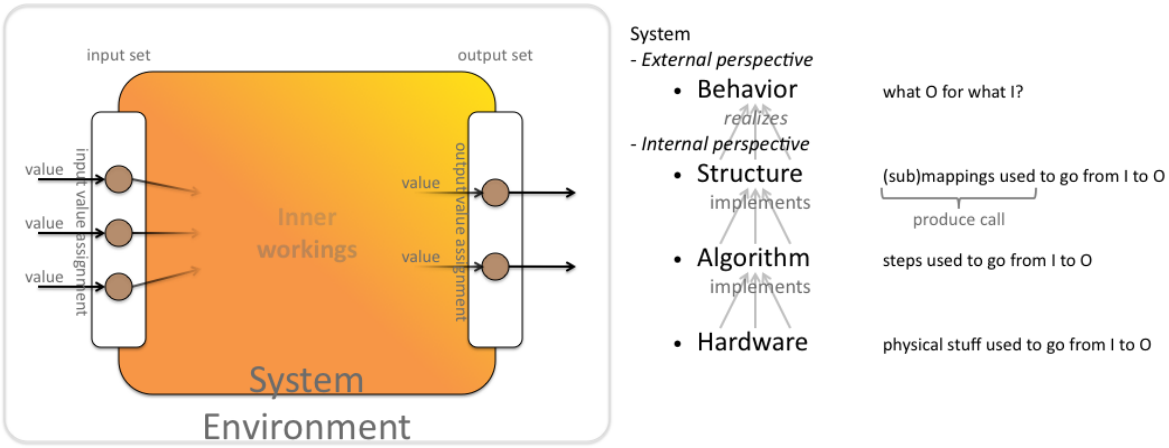


Figure 2: To the left: A graphical representation of a system in its environment. The environment presents the system (through its sensors) with values (an input value assignment) for its input set of input variables. This then goes through the inner workings of the system, which results in the system assigning values (an output value assignment) to its output set of output variables. These will, through the effectors of the system, in turn influence the environment. To the right: the different perspectives from and levels on which a system can be described discussed in this paper. From the external perspective, one looks at what the system does (its behavior), without considering its inner workings. The internal perspective on the other hand, focusses on the inner workings and describes how the system does what it does. This description can be in terms of (sub)mappings used (structure level), steps used (algorithmic level) and/or hardware used (hardware level). The arrows in the graph denote the relations between the different levels and perspectives.

kinds of responses to basic situations. The second element is a set of regulative layers that regulate the behavior of other layers to be appropriate for more complex aspects of the situation.

2. Representing system structure - conceptual structure

This paper is aimed at a formal comparison of Reactive, Hierarchical and Regulated Reactive Robotics. To that end, we require a formal and comparable description of those approaches. Since the approaches all describe the *structure* of systems[1], a structure-level description of systems could well fulfill that need.

However, we are not aware of any formalism for describing the structure-level of systems that does not assume a particular algorithmic implementation. Therefore, we will introduce such a formalism in this paper. The remainder of this section is devoted to conveying the intuitions behind the main concepts of this formalism and to relating the structure-level description of systems to other, more common, descriptions of systems. In Section 3 these intuitions will be formalized and in the remaining sections this formalism will be used to prove various statements about the hardware requirements of different systems structures, see Figure 3 for an overview.

2.1. Systems

Each robot is an embodied system, i.e. it is an entity (roughly) separable from its environment that produces output(value assignment)s when presented with input(value assignment)s. In more detail; the environment instills a particular activation on the sensors (input variables in the input set) of the system that then goes through the inner workings, resulting in a particular activation on the effectors (output variables in the output set) which in turn influences the environment[2] (see left of Figure 2).

[1] A reactive system has multiple distinct independent layers, an hierarchical system has sense-modules, act-modules and a central planner and a regulated reactive system is a reactive system with the addition of input-driven regulation. See as well Figure 1.

[2] This description being sequential is not intended to suggest that the process being described should be as well.

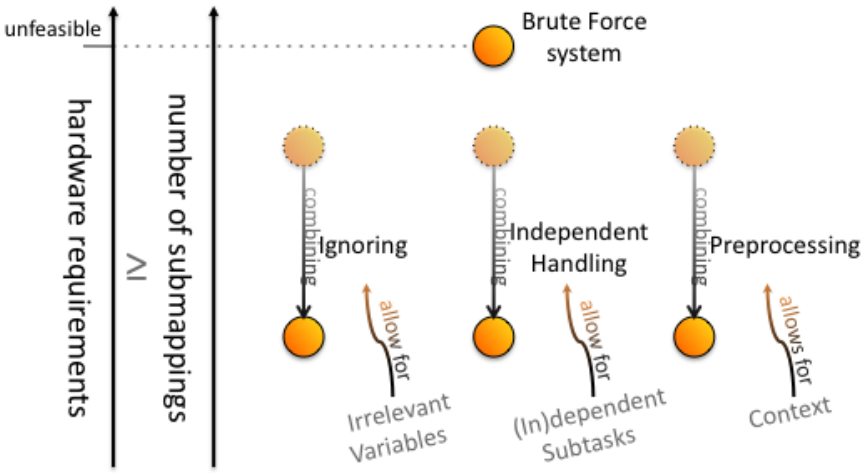


Figure 3: A graphical representation of the statements formally proven in this paper. The hardware requirements of a system are related to the number of submappings used to describe it on the structure level (Section 4). Since a brute force system requires unfeasibly many submappings, it thus as well requires unfeasibly much hardware (Section 5). Fortunately, we can improve on those requirements by combining submappings (Section 6). We discuss three conditions under which submappings can be combined effectively (irrelevant inputs, (in)dependent subtasks and context) (Section 7) and relate those to the different robotics approaches (Section 8).

One can describe a system from two perspectives, the internal and the external perspective. From the external perspective, one only looks at *what* the system does, i.e. what outputs are produced for what inputs. From the internal perspective, one looks at the inner workings of the system, *how* the system does what it does, i.e. what is going on inside the system when it is presented with an input. Furthermore, from the internal perspective one can describe a system with a particular level of implementational detail. One can describe it in terms of its hardware, generalize away from that and describe its algorithm or generalize even further and describe its structure (see right of Figure 2).

These levels of description of different systems have similarities to those first introduced by Marr [8] (but see as well [9]). One could say that, roughly, the computational level of Marr is our external level description of behavior. Similarly, the algorithmic and hardware level of Marr are comparable to our algorithmic level and hardware level description respectively. In terms of the levels of Marr, the structure level then would either be a special kind of algorithmic level description or a new level of description altogether. However, these are only rough equalities. Therefore we will now discuss our levels of description in more detail and stress relevant implicit properties thereof.

2.2. Levels of description for systems

Behavior One very common description of systems from the external perspective is that of behavior. Here, behavior will be taken to be a concise description of what outputs a system gives for what inputs[3].

Hardware If one wants to look at what is going on inside a system, i.e. from the internal perspective, one can look at its hardware. That is, how is the physical stuff the system exists of organized and how is a stimulation of the sensors physically transfered to an activation of the effectors. Physics is a very suitable tool to describe the hardware of a system and the interaction thereof. However, such a physical description of a system would rapidly become very extensive - consider for example how hard it would be to completely describe a robot (or a computer) and its behavior in terms of physics.

[3]It is becoming more and more common to speak of behavior not as just something that a system produces, but rather as (the result of) a relation between a system and its environment. Since we are here focusing on systems, we will use behavior more in the first sense. However, we wish to note that in our view behavior as it is used here is still compatible with the second sense.

Algorithm Thus, a more common description – that generalizes away from the particular implementational details of how it is physically realized – is that on the algorithmic level. On the algorithmic level one describes what 'steps' (or computations, if the system described is a computer or robot) are undertaken by the system. The specifics of the execution of these steps are ignored (which is why multiple hardware structures can implement the same algorithm) and instead one looks only at what these steps do. (Computational) complexity theory is the tool used to analyze algorithms and relate them to space and/or time requirements [REF]. Computational complexity theory is as well used to relate behaviors (and the (known) algorithms that can realize them) to space and/or time requirements [REF].

Structure However, sometimes even an algorithmic level description of a system can be too extensive. A commonly used, sofar informal, description that once more generalizes away, this time from the particular steps used, is that of structure. A relevant example is the characterization of a system as reactive or hierarchical. On the structure level, one describes the parts of a system, their specifications and their relation. We will describe this in more detail in the following section. In a fashion similar to that of computational complexity theory, we will try to relate structure level descriptions of systems to the space requirements of those systems.

We want to stress several implicit properties of these levels of description. First off, every behavior can be realized by multiple structures, algorithms and/or hardware configurations. We will give more extensive illustrations of this later on, but a trivial example is that the addition of a useless (in terms of producing behavior) "thing" to a structure/algorithm/hardware will not change its behavior – e.g. putting a sticker on a robot (in a convenient spot) does change its hardware, but need not notably change its behavior. Likewise, multiple hardware configurations can implement the same algorithm and multiple algorithms can implement the same structure.

On the other hand, each structure, algorithm and/or hardware configuration realizes only a single behavior. Likewise, each hardware configuration implements only one algorithm and each algorithm implements only one structure.

Furthermore, we believe it is important to realize that the different levels of description introduced here can in fact better be considered classes of levels of description. For example, a hardware level description can be mechanistic, but might be in quantum mechanical terms as well and an algorithm can have steps with various levels of abstraction (consider higher- and lower-order programming languages).

The last thing we want to stress is that all levels of description and perspectives of behavior are just that. That is, one can describe the behavior, structure, algorithm and hardware configuration of one and the same system, all at the same time, and those descriptions are in that way related. Thus, saying something on the structure of a system will also say something about the hardware configuration of a system, as it limits the possible hardware configurations to those – numerous as they may be – that implement algorithms that implement that structure and that thus realize the same behavior.

2.3. On the structure-level

We will here use structure in the following sense: A structure is a specification of parts[4] in terms of inputs distinguished and the relation between those parts.

We will define this 'specification of parts' by means of (sub)mappings and this specification of 'their relation' by means of a produce call.

Note that the different approaches to robotics are not structure level descriptions, but rather classes of structures. For example, Reactive Robotics defines the shared properties of the structure level descriptions of all reactive systems. More formally, we will take an approach to robotics to be the set of all systems that have the properties that characterize that approach.

[4]By using the word 'parts' here, we try to avoid the connotations that go with commonly used words such as components or modules. Rather, 'parts' as it is used here can among others refer to the distinct layers of a reactive system, the central planner of a hierarchical system and the input-driven regulation of a regulated reactive system.

3. Representing system structure - formal definitions

We will formalize the structure-level description of systems in terms of the different input value assignments (Section 3.1) that are distinguished by the system and associated with a response. To that end we will introduce submappings (Section 3.2) which are activated by a particular set of input value assignments – where the wildcard symbol will be used to represent that a submapping generalizes over the input values for a particular input variable.

Furthermore, we will introduce the produce call (Section 3.3), which describes how these submappings are related to one another as they are used to produce output value assignments.

3.1. Variables and value assignments

We will be talking about **variables** (denoted by v), which can be among others **input variables** (denoted by v^I) or **output variables** (denoted by v^O). Each variable v' has a **domain** (denoted by $D_{v'}$), which is a set of values that variable v' can take. All domains contain the **wildcard** value (denoted by *?*) ($\forall_{v'}[?{\in}D_{v'}]$). A variable set (denoted by V) is a set of variables, each of which can have a different domain. The domain of a variable set V' (denoted by $D_{V'}$) is the union of the domain of all variables in it ($D_{V'} = \cup_{v' \in V'}[D_{v'}]$). A variable set containing only input variables or output variables is an **input set** (denoted by V^I) or **output set** (denoted by V^O) respectively.

A **value assignment** for a variable set V' (denoted by $\underline{w}_{V'}$) is a function $\underline{w}_{V'}$: $V' \mapsto D_{V'}$ such that $\forall_{v' \in V'}[\underline{w}_{V'}(v'){\in}D_{v'}]$. A value assignment for an input set or output set is an **input value assignment** (denoted by $\underline{w}_{V^I}$) or **output value assignment** (denoted by $\underline{w}_{V^O}$) respectively. The **value assignment set** for a variable set V' (denoted by $W_{V'}$) is the set of all possible value assignments for that V' ($W_{V'} = \{\underline{w}_{V'} \mid\}$).

A value assignment $\underline{w}_V$ is said to be **complete** if it does not assign the wildcard value ($\forall_{v' \in V}[\underline{w}_{V'}(v') \neq ?]$). In turn, a value assignment set for variable set V' is said to be **complete** (denoted by $W^C{}_{V'}$) if it contains only complete value assignments ($\forall_{\underline{w}'_{V'} \in W^C{}_{V'}}[\forall_{v' \in V'}[\underline{w}_{V'}(v') \neq ?]]$).

3.1.1. Comparing value assignments

Due to the existence of wildcards, value assignments can be compared in three ways. They can be completely equal if they assign the same value to all variables they are defined for. They can overlap if they assign the same value or a wildcard value to all variables they are defined for. And one value assignment can generalize another value assignment if it assigns the same value or a wildcard value to all variables the other value assignment is defined for.

. The **equals** function $\underline{E}$: $W_V \times W_V \times 2^V \mapsto \{true, false\}$ is defined on two value assignments, $\underline{w^1}_V$ and $\underline{w^2}_V \in W_V$ for the same variable set V and a variable set V' subset of V, such that

$$\underline{E} := \begin{cases} true & iff\ \forall_{v' \in V'}[\underline{w^1}_V(v') = \underline{w^2}_V(v')] \\ false & otherwise \end{cases}$$

We will use $\underline{E}(\{\underline{w^1}_V, \underline{w^2}_V, \underline{w^3}_V, ...\}, V')$ as a shorthand for $\underline{E}(\underline{w^1}_V, \underline{w^2}_V, V') \wedge \underline{E}(\underline{w^2}_V, \underline{w^3}_V, V') \wedge \underline{E}(\underline{w^1}_V, \underline{w^3}_V, V') \wedge ...$ (i.e. equals yields *true* for all combinations of the given value assignments). As we will later show (Lemma 1) this will yield the same result, regardless of the order in which those functions are called.

. The **overlaps** function $\underline{O}$: $W_V \times W_V \times 2^V \mapsto \{true, false\}$ is defined on two value assignments, $\underline{w^1}_V$ and $\underline{w^2}_V \in W_V$ for the same variable set V and a variable set V' subset of V, such that

$$\underline{O} := \begin{cases} true & iff\ \forall_{v' \in V'}[\underline{w^1}_V(v') = \underline{w^2}_V(v')\ or\ \underline{w^1}_V(v') = ?\ or\ \underline{w^2}_V(v') = ?] \\ false & otherwise \end{cases}$$

We will use $\underline{O}(\{\underline{w^1}_V, \underline{w^2}_V, \underline{w^3}_V, ...\}, V')$ as a shorthand for $\underline{O}(\underline{w^1}_V, \underline{w^2}_V, V') \wedge \underline{O}(\underline{w^2}_V, \underline{w^3}_V, V') \wedge \underline{O}(\underline{w^1}_V, \underline{w^3}_V, V') \wedge ...$ (i.e. overlaps yields *true* for all combinations of the given value assignments). As we will later show (Lemma 2) this will yield the same result, regardless of the order in which those functions are called.

. The **generalizes** function $\underline{G}$: $W_V \times W_V \times 2^V \mapsto \{true, false\}$ is defined on two value assignments, $\underline{w^1}_V$ and $\underline{w^2}_V \in W_V$ for the same variable set V and a variable set V' subset of V, such that

$$\underline{G} := \begin{cases} true & iff\ \forall_{v' \in V'} [\underline{w^1}_V(v') = \underline{w^2}_V(v')\ or\ \underline{w^1}_V(v') = ?] \\ false & otherwise \end{cases}$$

3.1.2. Properties of the equals and overlaps function

For the equals and overlaps function, the order of the arguments is irrelevant (Lemma 1 and 2).

Lemma 1. $\underline{E}(\underline{w^1}_V, \underline{w^2}_V, V') = \underline{E}(\underline{w^2}_V, \underline{w^1}_V, V')$

Proof. trivial □

Lemma 2. $\underline{O}(\underline{w^1}_V, \underline{w^2}_V, V') = \underline{O}(\underline{w^2}_V, \underline{w^2}_V, V')$

Proof. The conditions for overlaps will yield the same result regardless of the order of the arguments $\underline{w^1}_V$ and $\underline{w^2}_V$, since $\underline{w^1}_V(v') = \underline{w^2}_V(v')$ is equal to $\underline{w^2}_V(v') = \underline{w^1}_V(v')$ and for each variable in V', both value assignments are checked for having the wildcard value. □

3.1.3. Combining value assignments

Here we introduce a function that can merge different overlapping value assignments such that as many wildcards as possible are replaced by non-wildcard values. This function will later on be used to merge multiple overlapping value assignments with wildcards into a single value assignment with less (or no) wildcards.

We as well introduce a function that can unite different value assignments for different value sets into a value assignment for the union of those sets. This function will later on be used to unite multiple 'partial' input and output value assignments into a single input or output value assignments.

Merging value assignments. The **merging** function $\underline{M}$: $W_V \times W_V \times 2^V \mapsto \{true, false\}$ is defined on a variable set V' subset of V ($V' \subseteq V$) and two value assignments, $\underline{w^1}_V$ and $\underline{w^2}_V \in W_V$, for the same variable set V, overlapping on V' ($\underline{O}(\underline{w^1}_V, \underline{w^2}_V, V') = true$) to yield a value assignment $\underline{w^\star}_V$ such that

$$\forall_{v \in V} [\underline{w^\star}_V(v)] := \begin{cases} ? & iff \neg\ v \notin V' \\ \underline{w^1}_V(v) & iff\ v \in V'\ and\ \underline{w^1}_V(v) \neq ?\ and\ \underline{w^2}_V(v) = ? \\ \underline{w^2}_V(v) & iff\ v \in V'\ and\ \underline{w^2}_V(v) \neq ?\ and\ \underline{w^1}_V(v) = ? \\ \underline{w^1}_V(v) & iff\ v \in V'\ and\ \underline{w^1}_V(v) = \underline{w^2}_V(v) \neq ? \\ ? & iff\ v \in V'\ and\ \underline{w^1}_V(v) = \underline{w^2}_V(v) = ?^{5} \end{cases}$$

We will use $\underline{M}(\{\underline{w^1}_V, \underline{w^2}_V, ..., \underline{w^n}_V\}, V')$ as a shorthand for $\underline{M}(\underline{w^1}_V, \underline{M}(\underline{w^2}_V, ...\underline{M}(\underline{w^{n-1}}_V, \underline{w^n}_V, V')..., V'), V')$. As we will later show (see Lemma 3, 4 and 5 below), this is allowed and will yield the same result regardless of the order in which those merges are ordered – provided that $\underline{O}(\{\underline{w^1}_V, \underline{w^2}_V, \underline{w^3}_V, ...\}, V') = true$.

Here $\underline{M}(\{\underline{w'}_V\}, V')$ will be treated as a special case and yield $\underline{M}(\{\underline{w'}_V, \underline{w'}_V\}, V')$.

Here $\underline{M}(\{\}, V')$ will be treated as a special case and yield a $\underline{w'}_V$ that assigns to all variables the wildcard value $?$.

Uniting value assignments. The **unite partial value assignments** function $\underline{MA}$: $W_{V^1} \times W_{V^2} \mapsto W_{V^1 \cup V^2}$ is defined two value assignments, $\underline{w^1}_{V^1} \in W_{V^1}$ and $\underline{w^2}_{V^2} \in W_{V^2}$, for non-overlapping variable sets V^1 and V^2 ($V^1 \cap V^2 = \emptyset$) to yield a value assignment $\underline{w^\star}_{V^1 \cup V^2}$ such that

$$\forall_{v' \in V^1 \cup V^2} [\underline{w^\star}_{V^1 \cup V^2}(v')] := \begin{cases} \underline{w^1}_{V^1}(v') & iff\ v' \in V^1 \\ \underline{w^2}_{V^2}(v') & iff\ v' \in V^2 \end{cases}$$

[5] We did not include conditions in which $v \in V'$, $\underline{w^1}_V(v) \neq \underline{w^2}_V(v)$, $\underline{w^1}_V(v) \neq ?$ and $\underline{w^2}_V(v) \neq ?$ since a condition for applying the function is that $\underline{w^1}_V$ and $\underline{w^2}_V$ overlap on V', which implies such conditions will never occur.

3.1.4. Properties of the merging function

Lemma 3. *(symmetry)* $V' \subseteq V \wedge \underline{O}(\underline{w^1}_V, \underline{w^2}_V, V') = true \rightarrow \underline{E}(\underline{M}(\underline{w^1}_V, \underline{w^2}_V, V'), \underline{M}(\underline{w^2}_V, \underline{w^1}_V, V'), V) = true$

Proof. For each variable v' not in V', both mergings will result in a function that assigns $?$ to v'. For each variable v' in V', since the value assignments $\underline{w^1}_V$ and $\underline{w^2}_V$ overlap on those variables, we have three options;

1. both have the same non-wildcard value, in which case merging will result in a function that assigns that value to v',
2. one of them has the wildcard value, the other a non-wildcard value, in which case merging will result in a function that assigns the non-wildcard value to v',
3. both of them have the wildcard value, in which case merging will result in a function that assigns $?$ to v'.

Consequently, both mergings will result in value assignments that assign the same value to all variables in V, i.e. that are equal. □

Lemma 4. *(transitivity)* $V' \subseteq V \wedge \underline{O}(\{\underline{w^1}_V, \underline{w^2}_V, \underline{w^3}_V\}, V') = true \rightarrow \underline{O}(\underline{M}(\underline{w^1}_V, \underline{w^2}_V, V'), \underline{w^3}_V, V') = true$

Proof. As follows from the definition, the merging will result in a value assignment $\underline{w^\star}_V$ that assigns to every variable either $?$ or the same value as $\underline{w^1}_V$ and/or $\underline{w^2}_V$ do. Since $\underline{w^1}_V$ and $\underline{w^2}_V$ were assumed to overlap with $\underline{w^3}_V$ on V', we know that for the variables v in both cases (i.e. for all variables) in V', $\underline{w^\star}_V(v) = \underline{w^3}_V(v)$ or $\underline{w^\star}_V(v) = ?$ or $\underline{w^3}_V(v) = ?$, i.e. that $\underline{w^\star}_V$ and $\underline{w^3}_V$ overlap on V'. □

Lemma 5. $V' \subseteq V \wedge \underline{O}(\{\underline{w^1}_V, \underline{w^2}_V, \underline{w^3}_V\}, V') = true \rightarrow \underline{E}(\underline{M}(\underline{M}(\underline{w^1}_V, \underline{w^2}_V, V'), \underline{w^3}_V, V'), \underline{M}(\underline{M}(\underline{w^1}_V, \underline{w^3}_V, V'), \underline{w^2}_V, V'), V) = true$

Proof. For each variable v' not in V', both compared mergings will result in a value assignment that assigns $?$ to v'. For each variable v' in V', since the value assignments $\underline{M}(\underline{w^1}_V, \underline{w^2}_V, V')$, $\underline{w^3}_V$, $\underline{M}(\underline{w^1}_V, \underline{w^3}_V, V')$ and $\underline{w^2}_V$ overlap (Lemma 4) on those variables, we have three options;

1. both have the same non-wildcard value, in which case the compared mergings will both result in a function that assigns that value to v',
2. one of them has the wildcard value, the other a non-wildcard value, in which case the compared mergings will both result in a function that assigns the same non-wildcard value to v',
3. both of them have the wildcard value, in which case the compared mergings will both result in a function that assigns $?$ to v.

Consequently, both compared mergings will result in value assignments that assign the same value to all variables in V, i.e. that are equal. □

3.2. Submappings and mappings

Here we will introduce submappings and mappings. Mappings consist of submappings, each of which is used to describe a situation that is distinguished and the output value assignment that is associated with that situation. They will, later on (Subsection 3.3), be used to produce an output value assignment for an input value assignment.

3.2.1. Submappings

A **submapping** for an input set V^I and an output set V^O (denoted by m_{V^I,V^O}) is a pair of an input value assignment for V^I and an output value assignment for V^O ($m_{V^I,V^O} = (\underline{w}_{V^I}, \underline{w'}_{V^O})$). Note that for m_{V^I,V^O}, $|V^I|$ need not be equal to $|V^O|$. To ease notation, for a submapping $m_{V^I,V^O} = (\underline{w}_{V^I}, \underline{w'}_{V^O})$ we define $m_{V^I,V^O}{}^I := \underline{w}_{V^I}$ and $m_{V^I,V^O}{}^O := \underline{w'}_{V^O}$.

A submapping can have one or more of the following properties;

- it is **basic** if its input value assignment is complete ($m_{V^{I\prime},V^{O}}{}^{I} \in W^{C}{}_{V^{I\prime}}$).
- it is **partial** if its output value assignment is not complete ($m_{V^{I\prime},V^{O}}{}^{O} \notin W^{C}{}_{V^{O\prime}}$) .
- it is **empty** if its output value assignment assigns only the wildcard value ($\forall_{v' \in V^{O}} [m_{V^{I},V^{O}}{}^{O}(v') = ?]$).

3.2.2. Mappings

A **mapping** for an input set V^I and an output set V^O (denoted by M_{V^I,V^O}) is a finite set of submappings for that input set and output set ($M_{V^I,V^O} = \{m^1{}_{V^I,V^O}, m^2{}_{V^I,V^O}, m^3{}_{V^I,V^O}, ...\}$) such that $\forall_{m'{}_{V^I,V^O},\, m''{}_{V^I,V^O} \in M_{V^I,V^O}} [\underline{O}(m'{}_{V^I,V^O}{}^{I}, m''{}_{V^I,V^O}{}^{I}, V^I) \rightarrow \underline{O}(m'{}_{V^I,V^O}{}^{O}, m''{}_{V^I,V^O}{}^{O}, V^O)]$. This condition, to which we will refer as the **consistency condition** will prevent conflicting output value assignments when we use mappings to produce output value assignments later on. Note that any set of submappings for the same input and output set to which the consistency condition applies is a mapping (including the empty set).

A mapping $M_{V^I,V^O} = \{m^1{}_{V^I,V^O}, m^2{}_{V^I,V^O}, m^3{}_{V^I,V^O}, ...\}$ with $V^I = \{v^{I1}, v^{I2}, ...\}$ and $V^O = \{v^{O1}, v^{O2}, ...\}$ can be represented as a table as follows;

v^{I1}	v^{I2}	...	v^{O1}	v^{O2}	...
$m^1{}_{V^I,V^O}{}^{I}(v^{I1})$	$m^1{}_{V^I,V^O}{}^{I}(v^{I2})$	...	$m^1{}_{V^I,V^O}{}^{O}(v^{O1})$	$m^1{}_{V^I,V^O}{}^{O}(v^{O2})$	...
$m^2{}_{V^I,V^O}{}^{I}(v^{I1})$	$m^2{}_{V^I,V^O}{}^{I}(v^{I2})$	...	$m^2{}_{V^I,V^O}{}^{O}(v^{O1})$	$m^2{}_{V^I,V^O}{}^{O}(v^{O2})$	...
$m^3{}_{V^I,V^O}{}^{I}(v^{I1})$	$m^3{}_{V^I,V^O}{}^{I}(v^{I2})$	...	$m^3{}_{V^I,V^O}{}^{O}(v^{O1})$	$m^3{}_{V^I,V^O}{}^{O}(v^{O2})$	...
...	...	...	...	...	...

We will refer to the input set and output set a submapping or Mapping $M_{V^{I\prime},V^{O\prime}}$ is for as its **environment** ($(V^{I\prime}, V^{O\prime})$).

3.2.3. Uniting mappings

We here introduce three notions (extending a submapping, partial mappings and uniting partial mappings) that will later on be used to describe how two mappings for non-overlapping input sets and output sets can be united into a single mapping for the union of those input sets and output sets.

A submapping $m'{}_{V^{I\prime},V^{O\prime}}$ is **extended** to an input set V^I such that $V^{I\prime} \subseteq V^I$ and an output set V^O such that $V^{O\prime} \subseteq V^O$ by extending its input value assignment and output value assignment such that they assign to all variables in $V^I \setminus V^{I\prime}$ and in $V^O \setminus V^{O\prime}$ the wildcard value.

The **unite partial mappings** function $\underline{\oplus}$: $\{M_{V^{I1},V^{O1}} |\} \times \{M_{V^{I2},V^{O2}} |\} \mapsto \{M_{V^{I1}\cup V^{I2},V^{O1}\cup V^{O2}} |\}$ is defined on two Mappings $M^1{}_{V^{I1},V^{O1}} \in \{M_{V^{I1},V^{O1}}\}$ and $M^2{}_{V^{I2},V^{O2}} \in \{M_{V^{I2},V^{O2}}\}$ where $V^{O1} \cap V^{O2} = \emptyset$[6] to yield the union of the submappings in $M^1{}_{V^{I1},V^{O1}}$ and $M^2{}_{V^{I2},V^{O2}}$ all extended to $V^{I1} \cap V^{I2}$ and $V^{O1} \cap V^{O2}$.

Because the output sets of the mappings combined were enforced not to overlap, the output value assignment of the submappings in the one mapping will always overlap those in the other. Thus the consistency condition will be satisfied in the united mapping.

We will refer to a set of mappings that can be merged into another mapping as **partial mappings** of that mapping.

3.2.4. Cutting mappings

We here introduce two notions (cropping a submapping and cutting a mapping) that will later on be used to create such partial mappings.

A submapping $m'{}_{V^I,V^O}$ can be **cropped** to an input set $V^{I\prime}$ such that $V^{I\prime} \subseteq V^I$ and an output set $V^{O\prime}$ such that $V^{O\prime} \subseteq V^O$ if its input value assignment assigns the wildcard value to all variables in V^I not in $V^{I\prime}$ ($\forall_{v^{I\prime} \in V^I \setminus V^{I\prime}} [m'{}_{V^I,V^O}{}^{I}(v^{I\prime}) = ?]$) and its output value assignment assigns the wildcard value to all variables in V^O not in $V^{O\prime}$ ($\forall_{v^{O\prime} \in V^O \setminus V^{O\prime}} [m'{}_{V^I,V^O}{}^{O}(v^{O\prime}) = ?]$). This cropping results in a new submapping $m^\star{}_{V^{I\prime},V^{O\prime}}$

[6] This might be overly restrictive, but will suffice for our purposes here.

that assigns to each variable in $V^{I\prime}$ and $V^{O\prime}$ the same value as m'_{V^I,V^O} does ($\forall_{v^{I\prime}\in V^{I\prime}}[m^\star_{V^{I\prime},V^{O\prime}}{}^I(v^{I\prime}) = m'_{V^I,V^O}{}^I(v^{I\prime})] \wedge \forall_{v^{O\prime}\in V^{O\prime}}[m^\star_{V^{I\prime},V^{O\prime}}{}^O(v^{O\prime}) = m'_{V^I,V^O}{}^O(v^{O\prime})]$).

The **cut** function $\underline{X}$: $\{M_{V^I,V^O}\ |\} \times 2^{V^I} \times 2^{V^O} \mapsto \{M_{V^{I\prime},V^{O\prime}}\ |\}$ is defined on a Mappings $M'_{V^I,V^O} \in \{M_{V^I,V^O}\ |\}$ and subsets of the input set and output set $V^{I\prime}$ and $V^{O\prime}$ to yield a mapping $M^\star_{V^{I\prime},V^{O\prime}} \in \{M_{V^{I\prime},V^{O\prime}}\ |\}$ that is the set of all submappings in M'_{V^I,V^O} that can be cropped to $V^{I\prime}$ and $V^{O\prime}$, cropped to $V^{I\prime}$ and $V^{O\prime}$.

Notice that the cropping will only "remove" assignments of wildcards and that a cropped submapping can thus easily be extended. As a result cropping and extending can be used to undo each other's effect. Likewise, cutting a mapping could be used to create partial mappings that can be merged into a mapping once more. Observe that if for all submappings in that mapping there is a cropped mapping in those partial mappings, this merging will result in the original mapping.

3.3. Producing output value assignments with a mapping

With our formal definition of mappings in place, we can now introduce the produce function. The produce function will by means of mappings get an output value assignment on the basis of an input value assignment. As we will use the combination of the produce function and its arguments as the structure level description of systems, we will only define it on complete input value assignments (since wildcards are intended for internal use)

3.3.1. Activating mappings

To define the produce function, we first need to know which submappings are relevant in responding to an input value assignment. The activated function will serve that purpose.

The **activated** function $\underline{A}$: $\{m_{V^I,V^O}\ |\} \times W^C{}_{V^I} \mapsto \{true, false\}$ is defined on a submapping $m'_{V^I,V^O} \in \{m_{V^I,V^O}\ |\}$ and a complete (i.e. containing no wildcards)[7] input value assignment, $\underline{w}'_{V^I} \in W^C{}_{V^I}$ such that

$$\underline{A} := \begin{cases} true & \textit{iff } \underline{O}(m'_{V^I,V^O}{}^I,\ \underline{w}'_{V^I},\ V^I) \\ false & otherwise \end{cases}$$

A submapping (m'_{V^I,V^O}) **distinguishes** between all input value assignments that activate it ($\forall_{\underline{w}'_{V^I}}[\underline{A}(m'_{V^I,V^O}{}^I,\ \underline{w}'_{V^I})=true]$) and those that do not ($\forall_{\underline{w}'_{V^I}}[\underline{A}(m'_{V^I,V^O}{}^I,\ \underline{w}'_{V^I})=false]$). A mapping whose input value assignment contains more wildcards, will overlap with and thus be activated by more input value assignments, which effectively leads to it doing less specific (or more generalized) distinguishing.

3.3.2. Using a mapping to produce output value assignments

The **produce** function $\underline{P}$: $\{M_{V^I,V^O}\ |\} \times W^C{}_{V^I} \mapsto W_{V^O}$ is defined on a mapping $M'_{V^I,V^O} \in \{m_{V^I,V^O}\ |\}$ and a complete input value assignment, $\underline{w}'_{V^I} \in W^C{}_{V^I}$ such that

$$\underline{P} := \underline{M}(\{m'_{V^I,V^O}{}^O \textit{ of a } m'_{V^I,V^O} \in M'_{V^I,V^O} \mid \underline{A}(m'_{V^I,V^O},\ \underline{w}'_{V^I}) = true\},\ V^O)$$

Or, in more natural language, a mapping produces for an input value assignment an output value assignment that is the combination (merging) of the output value assignments of all submappings in that mapping that were activated by the given input value assignment. Note that the consistency condition enforces that all activated mappings have overlapping output value assignments, which satisfies the condition for the merging function and thus allows for it being used here.

[7] This is because that input value assignment is something that is presented to the algorithm by its environment, while wildcards are intended for internal use. If one interprets wildcards as unassigned or unknown values, it would as well be quite odd to present an algorithm with an input value assignment that contains wildcards.

3.4. The behavior and structure of a system

With a produce function that has particular arguments (among which a mapping) we can appropriately describe how an output value assignment is produced when an input value assignment is presented in terms of situations distinguished and structure. We can as well describe behavior as a function on complete input value assignments that yields complete output value assignments.

In this section we will define these notions more formally, coming to a formal description of the behavior perspective on a system as well as the structure perspective on a system. These formal descriptions will then be related, much like in Figure 2.

3.4.1. A formal definition of systems

With the terminology in place, we can now define a system as an input set, an output set and "inner workings" such that for certain complete input value assignments to that input set it gives output value assignments to that output set. More concise descriptions of these inner workings can be given at the different levels of the internal perspective (e.g. on the structure, algorithm, or hardware level).

3.4.2. A formal definition of behavior

We define the **behavior** for an input set $V^{I\prime}$ and an output set $V^{O\prime}$ as a function $\underline{Beh_{V^{I\prime},V^{O\prime}}}: 2^{W^C{}_{V^{I\prime}}} \mapsto 2^{W^C{}_{V^{O\prime}}}$ that is defined on a complete input value assignment $\underline{w}'_{V^{I\prime}}$ from a subset of the set of all complete input value assignments to yield a complete output value assignment $\underline{w}'_{V^{O\prime}}$ from a subset of the set of all complete output value assignments.

These subsets of complete input and output value assignments on which a behavior is defined are the **input domain** (denoted by $Dom^I{}_{\underline{Beh_{V^{I\prime},V^{O\prime}}}}$) and **output domain** (denoted by $Dom^O{}_{\underline{Beh_{V^{I\prime},V^{O\prime}}}}$) of that behavior. Notice that all these value assignments are necessarily complete as the wildcard was defined only for internal use and the behavior is the external perspective.

3.4.3. Realizing behavior with a produce call

Consider a produce function with all its arguments filled in except for an input value assignment for input set $V^{I\prime}$ that yields an output value assignment for output set $V^{O\prime}$. We can now pick the biggest set of complete input value assignments $W^{C\prime}{}_{V^{I\prime}}$ for which that produce function will produce a complete output value assignment. We define $W^{C\prime}{}_{V^{O\prime}}$ as the set of all complete output value assignment thus produced. Consequently, that produce function is a description of a system. Furthermore, that produce function realizes a behavior $\underline{Beh_{V^{I\prime},V^{O\prime}}}$ with input domain $W^{C\prime}{}_{V^{I\prime}}$ and output domain $W^{C\prime}{}_{V^{O\prime}}$.

Here it is important to notice that it is not just the mapping (or mappings) used that realizes this relation between aforementioned produce function and the behavior, nor is it just the produce function. Rather, the mappings used describe what input value assignments are distinguished and the way in which they are used in a particular produce function describes in what way those mappings are organized and 'connected' to one another.

To capture this, we define the **produce call** to be a produce function with all its arguments filled in except for an input value assignment for input set $V^{I\prime}$ that yields an output value assignment for output set $V^{O\prime}$ that produces for complete input value assignments from input value assignment set $W^{C\prime}{}_{V^{I\prime}}$ complete output value assignments from output value assignment set $W^{C\prime}{}_{V^{O\prime}}$. Such a produce call thus describes a system with input set $V^{I\prime}$ and output set $V^{O\prime}$ on the structure level. It also realizes a behavior with input domain $W^{C\prime}{}_{V^{I\prime}}$ and output domain $W^{C\prime}{}_{V^{O\prime}}$.

Since this effectively means that the arguments – except for the input value assignment – of a produce call are fixed and part of the way in which the produce call realizes the behavior, we will represent a produce call by underlining all these fixed parameters, as that is how we have been denoting functions. To refer to any produce call (without considering its arguments) for an input set $V^{I\prime}$ and an output set $V^{O\prime}$, we will use $\underline{P_{V^{I\prime},V^{O\prime}}}$.

Consider for example $\underline{P(M'_{V^I,V^O}, \underline{w}_{V^I})}$, which is a produce call for all complete input value assignments for which it produces a complete output value assignment. We will construct less trivial examples later on.

Observe that because a produce call realizes a behavior, it as well has an input domain and output domain.

3.4.4. Multiple structures realizing the same behavior

As we suggested in Figure 2, one can now observe that multiple different structures (formalized produce calls) can all realize the same behavior. This can easily be illustrated with the following trivial example. Assume that $\underline{P(M'_{V^I,V^O}, \underline{w}_{V^I})}$ realizes a particular behavior. If we now add an empty submapping m'_{V^I,V^O} to M'_{V^I,V^O} that submapping will not change the output value assignment produced by the produce call and thus $\underline{P(\{m'_{V^I,V^O}\} \cup M'_{V^I,V^O}, \underline{w}_{V^I})}$ will realize the same behavior.

Comparing the behavior of structures. Knowing this, we can compare different produce calls on if they realize the same (or a more extensive) behavior or not. To that end we define the differs function. The **differs from** function $\underline{D}$: $\{\underline{P_{V^{I\prime},V^{O\prime}}}\ |\} \times \{\underline{P_{V^{I\prime},V^{O\prime}}}\ |\} \mapsto \{true, false\}$ is defined on two produce calls $\underline{P^1_{V^{I\prime},V^{O\prime}}}$ and $\underline{P^2_{V^{I\prime},V^{O\prime}}} \in \{\underline{P_{V^{I\prime},V^{O\prime}}}\ |\}$ such that

$$\underline{D} := \begin{cases} true & iff\ \exists_{\underline{w}'_{V^I} \in Dom^I_{\underline{P^2_{V^{I\prime},V^{O\prime}}}}} [\underline{E}(\underline{P^1_{V^{I\prime},V^{O\prime}}}(\underline{w}'_{V^I}),\ \underline{P^2_{V^{I\prime},V^{O\prime}}}(\underline{w}'_{V^I}),\ V^O) = false] \\ false & otherwise \end{cases}$$

Or, in more natural language, a produce call is said not to differ from one another produce call if it produces for all (complete) input value assignments in the domain of that other produce call equal (complete) output value assignments, i.e. if it realizes the same (or more extensive) behavior.

Since the produce calls are only compared on complete input value assignments from the input domain of the second produce call, even if the function yields *true*, it might be that the input domain of the first produce call is in fact a superset of that of the second produce call. The function thus is not symmetric.

4. More submappings in a produce call, more hardware required

In the previous Section we have given a formal definition of a system, behavior and produce calls and shown that they are, in fact, all different sides of the same coin. In what follows we will use these relationships to show that (truly) extending the behavior of a system, equals extending the structure of a system, equals extending the hardware of a system – which is to be expected as they are all different descriptions of the same thing (a system).

Imagine a system described at the structure level by produce call $\underline{P_{V^I,V^O}}$ that uses a set of submappings M with the following properties;

- there are no submappings in M that could be removed without changing the behavior of $\underline{P_{V^I,V^O}}$ (i.e. there are no redundant submappings)
- the hardware only implements $\underline{P_{V^I,V^O}}$
- $\underline{P_{V^I,V^O}}$ has input domain $Dom^I_{\underline{P_{V^I,V^O}}}$

Now if we would want $Dom^I_{\underline{P_{V^I,V^O}}}$ to include a new input value assignment $\underline{w}'_{V^I}$ we have only the following options;

- $\underline{w}'_{V^I}$ is included in $Dom^I_{\underline{P_{V^I,V^O}}}$ without any changes to $\underline{P_{V^I,V^O}}$. This implies that $\underline{w}'_{V^I}$ was already in $Dom^I_{\underline{P_{V^I,V^O}}}$ and thus that it is not truly new.
- $\underline{w}'_{V^I}$ is included in $Dom^I_{\underline{P_{V^I,V^O}}}$ with changes to $\underline{P_{V^I,V^O}}$. These can be one of the following;
  - $\underline{P_{V^I,V^O}}$ is adapted into by another produce call that covers the new, extended input domain. However, changing the produce call is changing the structure of the system and would thus result in another system.

– $\underline{P_{V^I,V^O}}$ is extended by adding a new submapping to M that properly handles the new input value assignment $\underline{w'}_{V^I}$. Since $\underline{P_{V^I,V^O}}$ now does more than it did, it will require more hardware.

So, if one wants to add a submapping to a mapping in a produce call, which is necessary to (truly) extend the input and output domain (without changing (the structure of) that mapping), one will need more hardware.

5. Brute force approach as a bad baseline

A **brute force system** is a produce call $\underline{P(M'_{V^I,V^O}}, \underline{w'_{V^I}})$ such that M'_{V^I,V^O} contains only basic submappings ($\forall_{m'_{V^I,V^O} \in M'_{V^I,V^O}} [m'_{V^I,V^O} \textit{ is basic}]$).

Lemma 6. $\forall_{M'_{V^I,V^O}} [\forall_{m'_{V^I,V^O} \in M'_{V^I,V^O}} [m'_{V^I,V^O} \textit{ is basic}] \rightarrow |M'_{V^I,V^O}| \geq |Dom^I_{\underline{P(M'_{V^I,V^O}}, \underline{w'_{V^I}})}|]$

Proof. Since a basic submapping m'_{V^I,V^O} contains per definition no wildcards, it will only overlap and thus be activated by one particular input value assignment (the one equal to $m'_{V^I,V^O}{}^I$). Thus, there needs to be at least one basic submapping for each input value assignment in the domain $Dom^I_{\underline{P(M'_{V^I,V^O}}, \underline{w_{[I]'}})}$ in M_{V^I,V^O} to even have activated submappings. □

Note that we will need more than one submapping iff not all those submappings produce a complete output value all by themselves (i.e. if they are partial submappings), or if the submappings have overlapping input value assignments.

In real world situations, a system (such as a robot) has to be defined for all possible situations it can run into (in so far as they are distinguishable as different input value assignments). It is a logical impossibility be that a system does not behave in the real world (even doing nothing is a behavior). Thus, to work properly in real world situations only mappings whose input domain ($Dom^I_{\underline{P(M'_{V^I,V^O}}, \underline{w_{[I]'}})}$) contains all possible combinations of values for its input set are usable. The size of the input domain will then exponentially grow with the size of the input set ($|Dom^I_{\underline{P(M'_{V^I,V^O}}, \underline{w_{[I]'}})}| = \prod_{v^I \in V^I} [|D_{v^I}|]$).

Consequently, the number of submappings used by a brute force system will grow exponentially with the size of its input set (follows from above lemma) if it is to be applied to real world situations. As for realistic problems these input sets will be quite large, this (and thus a brute force system) would better be avoided to prevent excessive hardware requirements.

6. Combining submappings: a tool to reduce resource requirements

So, we know that for real-life problems with realistically sized input sets the use of only basic submappings requires unfeasible amounts of hardware. Hence, we are in dire need of non-basic mappings that still produce the same behavior. To this end, we have set up all of the above with wildcards. As it is used in most of the above definitions, assigning the wildcard value to a variable effectively implies that it will be ignored; it does not play a role in overlapping anymore, it is not given priority in merging and a produce uses both overlapping and merging. Consequently, the wildcard can be used to ignore particular differences between situations (input value assignments in the input domain) when determining how to handle them. This way, one can use a single submapping to handle multiple situations. In what follows we will flesh out this method and define conditions under which one can introduce such non-basic submappings.

6.1. Illustration of the method

Before we introduce a formal notion, we will first illustrate the method that we intend to use:

Assume we have a mapping $M_{\{a,b\},\{o\}}$ (where $\forall_{v \in V^I \cup V'^O} [D_v = \{0,1\}]$) (see Figure 4(a))

We can recognize that all submappings that are activated by an input value assignment of *0* to *a*, give the same output value assignment to *o* (both *0*), regardless of the input value assignment to *b*. So we could

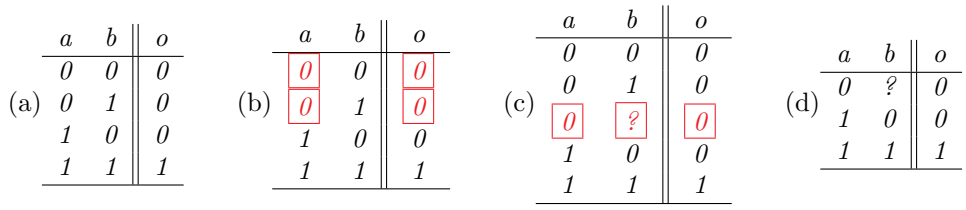


Figure 4: Illustration of how we will combine submappings. If we have a mapping (a), we can realize that particular variables can be generalized over (b) by the introduction of a new submapping with wildcards (c) that can replace the non-generalizing submappings originaly in the mapping (d), which could well reduce the size of the mapping.

replace those submappings by a single submapping that just ignores b (by giving it the wildcard value) (see Figure 4(d)).

A produce call using this mapping for an input value assignment of 0 to a will still yield the same result, but we have reduced the number of submappings by one.

We will formalize this as a two-step method. First the replacing submapping will be introduced into the mapping (see Table 4(a-c), Section 6.2). Second, all submappings that have as a consequence become empty will be removed (see Table 4(c-d), Section 6.3).

6.2. Combining submappings

The **combine** function $\underline{C}$: $\{M_{V^I,V^O} \mid\} \times \{m_{V^I,V^O} \mid\} \times 2^{V^O} \mapsto \{M_{V^I,V^O} \mid\}$ is defined on a mapping $M'_{V^I,V^O} \in \{M_{V^I,V^O} \mid\}$, a submapping $m'_{V^I,V^O} \in \{m_{V^I,V^O} \mid\}$ and an output set $V^{O\prime} \subseteq V^O$ with the following properties:

- Set of submappings $G = \{m''_{V^I,V^O} \in M'_{V^I,V^O} \mid \underline{G}(m'_{V^I,V^O{}^I},\ m''_{V^I,V^O{}^I},\ V^I) = true\}$
- $\forall_{m''_{V^I,V^O} \in G}[\underline{G}(m''_{V^I,V^O{}^O},\ m'_{V^I,V^O{}^O},\ V^{O\prime}) = true]$
- $\forall_{m''_{V^I,V^O} \in M'_{V^I,V^O}}[\underline{O}(m'_{V^I,V^O{}^I},\ m''_{V^I,V^O{}^I},\ V^I) = true \rightarrow \underline{O}(m'_{V^I,V^O{}^O},\ m''_{V^I,V^O{}^O},\ V^{O\prime}) = true]$

to yield a new mapping

$$\underline{C} := M'_{V^I,V^O} \cup \{m'_{V^I,V^O}\} \cup \{m''_{V^I,V^O} \mid m''_{V^I,V^O{}^I} = m''_{V^I,V^O{}^I},\ m''_{V^I,V^O{}^O} = \underline{M}(\{m''_{V^I,V^O{}^O}\}, V^O \setminus V^{O\prime}),\ m''_{V^I,V^O} \in G\} \setminus G$$

Or, in more natural language, we add to mapping M'_{V^I,V^O} the submappings m'_{V^I,V^O} and we replace each submapping in G by a new submapping that is different only in that it assigns $?$ to all variables in $V^{O\prime}$ (see Figure 4(c)).

6.3. Removing empty submappings

The **remove empty sumbmappings** function $\underline{R}$: $\{M_{V^I,V^O} \mid\} \mapsto \{M_{V^I,V^O} \mid\}$ is defined on a mapping $M'_{V^I,V^O} \in \{m_{V^I,V^O} \mid\}$ to yield a new mapping

$$\underline{R} := \{m'_{V^I,V^O} \in M'_{V^I,V^O} \mid \neg\ m'_{V^I,V^O}\ is\ empty\}$$

6.4. Properties of combining and removing

Lemma 7. $\underline{D}(\underline{P}(...M'_{V^I,V^O}...,\ \underline{w}_{[I]}),\ \underline{P}(...\underline{R}(M'_{V^I,V^O})...,\ \underline{w}_{[I]})) = false$

Proof. An empty submapping contributes nothing but wildcards to a merging. Since the end result of a produce is a merging, an empty submapping contributes nothing but wildcards to a produce either. Because a differs from checks if two mappings differ in the output value assignments they produce for input value assignments in the input domain of the first, removing all empty submappings will thus not cause a difference noted by a differs from. □

If a submapping is activated by an input value assignment, a submapping that generalizes that submapping will also be activated by that input value assignment:

Lemma 8. $\underline{G}(m^2{}_{V^I,V^O{}^I},\ m^1{}_{V^I,V^O{}^I},\ V^{I\prime}) \wedge \underline{A}(m^1{}_{V^I,V^O},\ \underline{w}'{}_{V^{I\prime}}) \rightarrow \underline{A}(m^2{}_{V^I,V^O},\ \underline{w}'{}_{V^{I\prime}})$

Proof. trivial □

Lemma 9. $\underline{D}(\underline{P}(...M'{}_{V^I,V^O}...,\ \underline{w}[_I]'),\ \underline{P}(...\underline{C}(M'{}_{V^I,V^O},\ m'{}_{V^I,V^O},\ V^{O\prime})...,\ \underline{w}[_I]')) = false$

Proof. By combining, in $M'{}_{V^I,V^O}$ a set of mappings G is replaced by a set of mappings that assign the wildcard value to all variables in $V^{O\prime}$ but are otherwise equal. The mapping $m'{}_{V^I,V^O}$ has an input value that generalizes that of all mappings in G and produces an output value that is generalized by that of all mappings in G. Hence, for all input value assignments in the input domain of $M'{}_{V^I,V^O}$ that the mappings in G are activated by, the replacing mappings will contribute to a produce the same output value assignment for all variables not in $V^{O\prime}$ while $m'{}_{V^I,V^O}$ will contribute to a produce the same output value assignment for all variables in $V^{O\prime}$ (Lemma 8).

Thus, combining will not cause a difference noted by a differs from. □

Lemma 10. $\underline{D}(\underline{P}(...M'{}_{V^I,V^O}...,\ \underline{w}[_I]'),\ \underline{P}(...\underline{R}(\underline{C}(M'{}_{V^I,V^O},\ m'{}_{V^I,V^O},\ V^{O\prime}))...,\ \underline{w}[_I]')) = false$

Proof. Follows trivially from Lemma 7 and 9 □

Observe that removing empty submappings reduces the size of a mapping. Observe that combining can result in empty submappings.

Consequently, by combining n times in the same mapping, resulting in a total of m submappings becoming empty submappings and then removing all empty submappings, if $m > n$, we can reduce the size of a mapping (as in the illustration of the method). Note that all this is done without differences in the behavior of the mapping arising, i.e. the same thing is being done with less resources.

So, combining and removing can be used to reduce the size of a mapping. Now all that is left to do, is find effective ways to structuraly combine in such a way that we end up with a mapping that is notably smaller. However, to do so more effectively, we will first introduce a function that allows us to efficiently combine multiple times with a single function call.

6.5. Combining over

To more efficiently use the combine function, we here introduce the combine over function which combines multiple combines into a single function call.

The **combine over** function $\underline{C_{over}}$: $\{M_{V^I,V^O}\ |\} \times 2^{V^I} \times 2^{V^O} \mapsto \{M_{V^I,V^O}\ |\}$ is defined on a mapping $M'{}_{V^I,V^O} \in \{m_{V^I,V^O}\ |\}$, an input set $V^{I\prime} \in 2^{V^I}$ and an output set $V^{O\prime} \in 2^{V^O}$ with the property that:

- $\forall_{m^1{}_{V^I,V^O},\ m^2{}_{V^I,V^O} \in M'{}_{V^I,V^O}} [\underline{O}(m^1{}_{V^I,V^O{}^I},\ m^2{}_{V^I,V^O{}^I},\ V^I) = true \rightarrow \underline{O}(m^1{}_{V^I,V^O{}^O},\ m^2{}_{V^I,V^O{}^O},\ V^{O\prime}) = true]$

Where G is the set of mappings $\{g^1, g^2, ..., g^n\}$ that consist of the mergings of the input value assignments over $V^{I\prime}$ and output value assignments over $V^{O\prime}$ for all biggest sets of mappings with overlapping input value assignments (and thus output value assignments)

$$
\begin{aligned}
G := \{\ & m_{V^I,V^O} \mid m_{V^I,V^O{}^I} = \underline{M}(\{m'{}_{V^I,V^O{}^I} \textit{ of a } m'{}_{V^I,V^O} \in M^\star{}_{V^I,V^O}\ |\},\ V^{I\prime}),\\
& m_{V^I,V^O{}^O} = \underline{M}(\{m'{}_{V^I,V^O{}^O} \textit{ of a } m'{}_{V^I,V^O} \in M^\star{}_{V^I,V^O}\ |\},\ V^{O\prime}),\\
& M^\star{}_{V^I,V^O} \in \{\ M''{}_{V^I,V^O} \subseteq M'{}_{V^I,V^O} \mid \forall_{m^1{}_{V^I,V^O},\ m^2{}_{V^I,V^O} \in M''{}_{V^I,V^O}} [\underline{O}(m^1{}_{V^I,V^O{}^I},\ m^2{}_{V^I,V^O{}^I},\ V^{I\prime}) = true],\\
& \quad \forall_{m^1{}_{V^I,V^O} \in M'{}_{V^I,V^O},\ m^2{}_{V^I,V^O} \in M''{}_{V^I,V^O}} [\underline{O}(m^1{}_{V^I,V^O{}^I},\ m^2{}_{V^I,V^O{}^I},\ V^{I\prime}) = true \rightarrow m^1{}_{V^I,V^O} \in M''{}_{V^I,V^O}]\}\}
\end{aligned}
$$

to yield a new mapping

$$\underline{C_{over}} := \underline{C}(\underline{C}(...\underline{C}(M'{}_{V^I,V^O},\ g^n,\ V^{O\prime}),...,\ g^2,\ V^{O\prime}),\ g^1,\ V^{O\prime})$$

Combining over will increase the size of a mapping by the number of distinct non-overlapping input value assignments for variables in V^I of submappings in it (since that is the size of G).

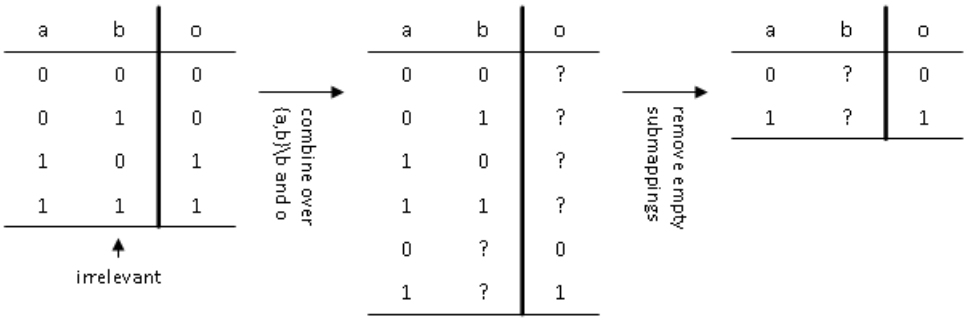


Figure 5: Example of the application of the combine function to a mapping with irrelevant variables. In the mapping to the left, variable b is irrelevant, and thus one can combine over the other variables which results in the mapping in the middle. Removing the empty submappings from that mapping results in the mapping to the right, which is much smaller than was the original mapping.

7. Conditions that allow for effective combining

In this section we will define various conditions (listed in Figure 3) on the environment that allow for effective combining of submappings in a produce call (i.e. such that the number of submappings can be significantly reduced). For each of the subsections that follows we will for each condition give a formal definition and an example of how it allows for combining, the results of which we then generalize to formal proofs of its effectiveness.

7.1. Irrelevant input variables

Not all variables in the input set of a mapping are necessarily of relevance for determining the correct output value assignment; they can be irrelevant as well. Consider for example the input value assignments a robot would receive from a broken (or useless) sensor. Such an irrelevant variable will reveal itself in the mapping because submappings that only differ on the value they assign to that variable will not have different output value assignments.

7.1.1. Formal definition

A set of input variables $V^{I\prime}$ subset of an input set V^I is said to be **irrelevant** to a mapping ${M'}_{V^I,V^O}$ if all two submappings ${m^1}_{V^I,V^O}$, ${m^2}_{V^I,V^O}$ in ${M'}_{V^I,V^O}$ that overlap on all not irrelevant input variables also overlap on the output set $(\underline{O}({m^1}_{V^I,V^O}{}^I,\ {m^2}_{V^I,V^O}{}^I,\ V^I \backslash V^{I\prime}) \rightarrow \underline{O}({m^1}_{V^I,V^O}{}^O,\ {m^2}_{V^I,V^O}{}^O,\ V^O))$.

7.1.2. Example

See Figure 5 for a graphical representation of the method we will use to reduce the size of a mapping by combining over irrelevant variables.

7.1.3. Results

As can be seen in the example, one can readily combine over irrelevant input variables, which results in quite a big reduction of the number of submappings (it is halved in the example with just one binary irrelevant input variable).

Combining over irrelevant input variables. From the definition of irrelevant variables follows that if $V^{I\prime}$ subset of V^I is irrelevant to mapping ${M'}_{V^I,V^O}$ we can readily combine over $V^I \backslash V^{I\prime}$ and V^O; $\underline{C_{over}}({M'}_{V^I,V^O},\ V^I \backslash V^{I\prime},\ V^O)$. Since this will replace all submappings in ${M'}_{V^I,V^O}$ by empty submappings (besides adding the new submappings), we can remove all empty submappings; $\underline{R}(\underline{C_{over}}({M'}_{V^I,V^O},\ V^I \backslash V^{I\prime},\ V^O))$. This will remove all submappings originally in ${M'}_{V^I,V^O}$ and leave us with only the added new mappings.

Reduction of size by combining over irrelevant input variables. Would we combine over a set of irrelevant input variables $V^{I\prime}$ as described above, in the mapping M'_{V^I,V^O} of a brute force system ($\underline{P}(\overline{M'_{V^I,V^O}}, \underline{w}_{[I]'})$) that is defined for all possible input value assignments to V^I ($|M'_{V^I,V^O}| = |Dom^I_{\underline{P}(\overline{M'_{V^I,V^O}}, \underline{w}_{[I]'})}| = \prod_{v^I \in V^I} \lceil |D_{v^I}| \rceil$), we would thus reduce the number of submappings needed (to $\prod_{v^I \in V^I \setminus V^{I\prime}} \lceil |D_{v^I}| \rceil$) by $\prod_{v^I \in V^{I\prime}} \lceil |D_{v^I}| \rceil$. This is an exponential reduction of size, just by ignoring irrelevant inputs.

7.1.4. Discussion

As we have shown, combining over irrelevant variables reduces the size of a mapping, while the behavior is unaffected. We will refer to this as **ignoring** those irrelevant variables, because that is effectively what happens.

Through showing that irrelevant variables can be ignored, we have also demonstrated the application and applicability of our methodology. The following conditions that allow for combining will be discussed in a similar sense.

7.2. (In)dependent subtasks

In a mapping it might well be that for determining the correct output value assignment for variables in a subset of the output set, only variables from a subset of the input set are of relevance. We will refer to the combination of aforementioned subsets of the input set and output set as a subtask. For example, the activation of the sound-sensor of a robot need not be relevant for one activity (such as picking things up) – even though it can be relevant for other activities (such as avoiding predators). Such a subtask will reveal itself in the mapping because submappings that only differ on the value they assign to variables not in the input set of a subtask will not have different output value assignments for variables in the output set of that subtask.

We will distinguish between two kinds of subtasks. In an independent subtask, none of the variables in the input set of that subtask is of relevance for determining the correct output value assignment for a variable not in the output set of that subtask. For a dependent subtask, the same is not true for a particular subset of the input set of that subtask. This way, an independent subtask represents a subtask that has nothing to do with any other subtask (hence the nomer independent), while there exists a dependency (through the particular subset) between a dependent subtask and other subtasks.

7.2.1. Formal definitions

A set of input variables $V^{I\prime}$ and output variables $V^{O\prime}$, subset of an input set V^I and output set V^O respectively, is said to be a **subtask** to a mapping M'_{V^I,V^O} if all two submappings $m^1{}_{V^I,V^O}$, $m^2{}_{V^I,V^O}$ in M'_{V^I,V^O}, that overlap on those input variables also overlap on those output variables ($\underline{O}(m^1{}_{V^I,V^O}{}^I, m^2{}_{V^I,V^O}{}^I, V^{I\prime}) \rightarrow \underline{O}(m^1{}_{V^I,V^O}{}^O, m^2{}_{V^I,V^O}{}^O, V^{O\prime})$).

In other words, a subtask consists of parts of the input and output set such that the rest of the input set is irrelevant in determining the correct values for that output set.

Independent subtasks. A subtask $V^{I\prime}$, $V^{O\prime}$ to a mapping M'_{V^I,V^O} is said to be an **independent subtask** if all two submappings $m^1{}_{V^I,V^O}$, $m^2{}_{V^I,V^O}$ in M'_{V^I,V^O}, that overlap on the input variables not in $V^{I\prime}$ overlap on the output variables not in $V^{O\prime}$ ($\underline{O}(m^1{}_{V^I,V^O}{}^I, m^2{}_{V^I,V^O}{}^I, V^I \setminus V^{I\prime}) \rightarrow \underline{O}(m^1{}_{V^I,V^O}{}^O, m^2{}_{V^I,V^O}{}^O, V^O \setminus V^{O\prime})$).

In other words, an independent subtask is a part of the input and output set such that the rest of the input set is irrelevant in determining the correct value for that output set and such that that input set is irrelevant in determining the correct value for the rest of the output set.

Observe that if for M'_{V^I,V^O} subtask $V^{I\prime}$, $V^{O\prime}$ is an independent subtask, then so is $V^I \setminus V^{I\prime}$, $V^O \setminus V^{O\prime}$.

Dependent subtasks. Two subtasks will be said to be dependent if they have overlapping input sets, but non-overlapping output sets.

Consider M'_{V^I,V^O} with subtasks V^{I1}, V^{O1} and V^{I2}, V^{O2} such that $V^{I1} \cup V^{I2} = V^I$ and $V^{O1} \cup V^{O2} = V^O$. If the input sets of those subtasks overlap on $V^{I\prime} = V^{I1} \cap V^{I2}$ and $V^{O1} \cap V^{O2} = \emptyset$, those subtasks are **dependent**.

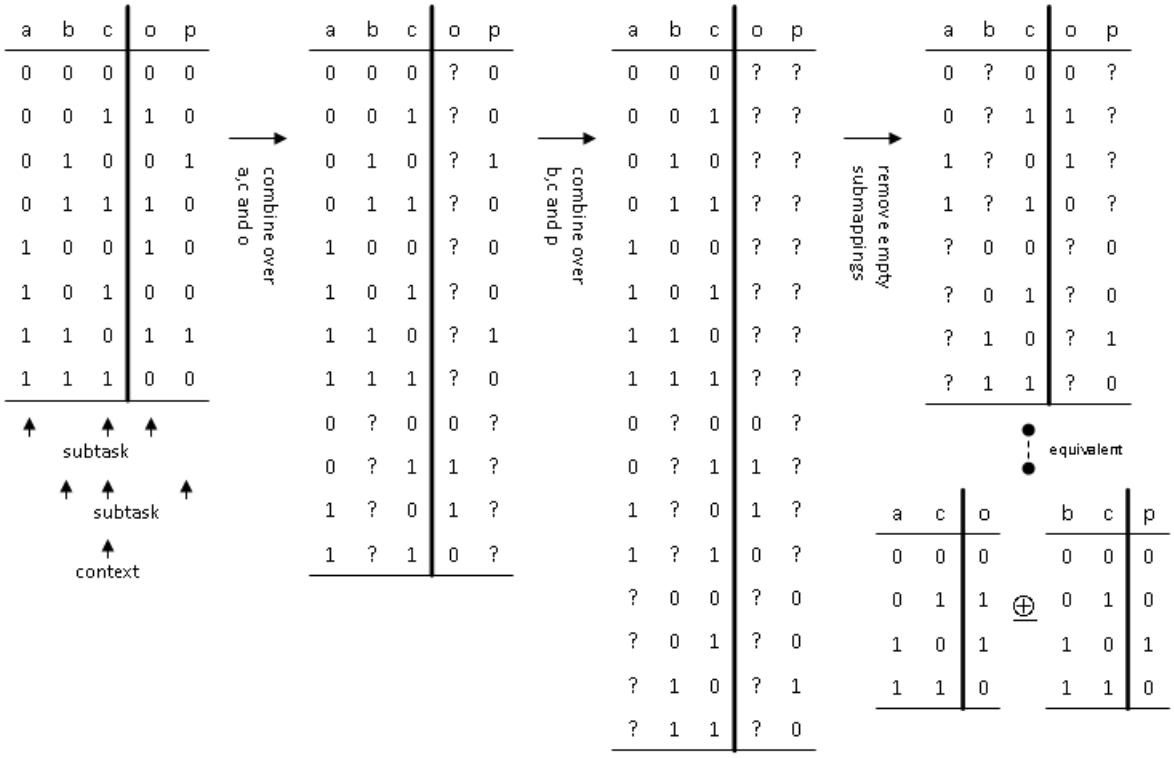


Figure 6: Example of the use of the partial mappings to handle dependent subtasks. In the mapping to the left, one can distinguish two dependent subtasks ($\{a,c\}$,$\{o\}$ and $\{b,c\}$,$\{p\}$). One can then combine over those two subtasks (as is done in the mappings in the middle). Removing the empty submappings from the resulting mapping yields the mapping on the topright, which could be much smaller than was the original mapping. This mapping in turn can be rerepresented as the two partial mappings on the bottomright.

Relation between independent and dependent subtasks. Note that if the overlap between dependent subtasks is the empty set, we are in fact talking about independent subtasks. Hence, the independent subtask is a special case of the dependent subtask, where the overlap is empty ($V^{I\prime} = \emptyset$). Consequently, in what follows we will focus on dependent subtasks (and then specify what our findings imply for independent subtasks).

7.2.2. Example

See Figure 6 for a graphical representation of the method we will use to reduce the size of a mapping by combining over dependent subtasks.

7.2.3. Results

As can be seen in the example, by combining over subtasks one can reduce the size of a mapping without affecting its behavior. In the following sections we will prove this more formally. Furthermore, we will rerepresent the resulting mapping as a pair of partial mappings, each performing one of the subtasks.

Combining over (in)dependent subtasks. From the definition of dependent subtasks follows that if we have mapping M'_{V^I,V^O} consisting only of dependent subtasks V^{I1}, V^{O1} and V^{I2}, V^{O2} and define $V^{I\prime} = V^{I1} \cap V^{I2}$, we can readily combine over these subtasks; $\underline{C_{over}}(\underline{C_{over}}(M'_{V^I,V^O}, V^{I1}, V^{O1}), V^{I2}, V^{O2})$. Since the mapping consists only of the two subtasks, this will replace all submappings originally in M'_{V^I,V^O} by empty submappings (besides adding the new submappings). A remove empty submappings $\underline{R}(\underline{C_{over}}(\underline{C_{over}}(M'_{V^I,V^O}, V^{I1}, V^{O1}), V^{I2}, V^{O2}))$ will thus remove all submappings originally in M'_{V^I,V^O} and leave us with only the added new mappings.

Reduction of size by combining over (in)dependent subtasks. Would we combine over two subtasks V^{I1}, V^{O1} and V^{I2}, V^{O2} with input sets overlapping in $V^{I\prime}$ as described above, in the mapping M'_{V^I, V^O} of a brute force system ($\underline{P}(M'_{V^I, V^O}, \underline{w}_{\lfloor I \rfloor'})$) that is defined for all possible input value assignments to V^I ($|M'_{V^I, V^O}| = |Dom^I_{\underline{P}(M'_{V^I, V^O}, \underline{w}_{\lfloor I \rfloor'})}| = \prod_{v^I \in V^I} \lfloor |D_{v^I}| \rfloor$), we would thus reduce the size to $\prod_{v^I \in V^{I\prime}} \lfloor |D_{v^I}| \rfloor * (\prod_{v^I \in V^{I1} \setminus V^{I\prime}} \lfloor |D_{v^I}| \rfloor + \prod_{v^I \in V^{I2} \setminus V^{I\prime}} \lfloor |D_{v^I}| \rfloor)$. This is a rather big reduction of size, even though it gets smaller as the overlap of the subtasks ($V^{I\prime}$) gets bigger[8]. Notice that if the two subtasks are independent, the overlap has minimal size ($V^{I\prime} = \emptyset$) and the reduction by combining over the subtasks will be maximal.

What is more, each of the submappings in this new mapping only has to determine the value for part of the output set (on the basis of only part of the input set), which will presumably reduce the resource requirements even further.

Rerepresenting a Mapping that has Combined over Subtasks. Consider the mapping resulting from combining over (in)dependent subtasks V^{I1}, V^{O1} and V^{I2}, V^{O2} in a mapping M'_{V^I, V^O} that consists only of those subtasks (such as the mapping in the top right corner of the example Figure 6). Because that mapping has been combined over both subtasks successively, it will contain only submappings that either have wildcards for V^{I1} and V^{O1} or wildcards for V^{I2} and V^{O2}. As wildcards effectively are ignored one can imagine that these two groups of submappings can be easily rerepresented as $M^1{}_{V^{I1}, V^{O1}}$ (by cutting over V^{I1} and V^{O1}, $\underline{X}(M'_{V^I, V^O}, V^{I1}, V^{O1})$) and $M^2{}_{V^{I2}, V^{O2}}$ (by cutting over V^{I2} and V^{O2}, $\underline{X}(M'_{V^I, V^O}, V^{I2}, V^{O2})$) respectively. These two mappings could then be united by using the unite partial mappings function to yield M'_{V^I, V^O} again.

Consider for example the mappings $M_{\{a,b,c\},\{o,p\}}$, $M_{\{a,c\},\{o\}}$ and $M_{\{b,c\},\{p\}}$ (see Figure 6). One can easily observe that $M_{\{a,b,c\},\{o,p\}} = \underline{\oplus}(M_{\{a,c\},\{o\}}, M_{\{b,c\},\{p\}})$. From this one can derive the following;

- $M_{\{a,c\},\{o\}}$ and $M_{\{b,c\},\{p\}}$ are partial mappings of $M_{\{a,b,c\},\{o,p\}}$
- We can use the produce function with the merged partial mappings as follows; $\underline{P}(\underline{\oplus}(M_{\{a,c\},\{o\}}, M_{\{b,c\},\{p\}}), \underline{w'}_{\{a,b\}})$
- Which will thus have the same result as using $M_{\{a,b,c\},\{o,p\}}$; $\underline{D}(\underline{P}(\underline{\oplus}(M_{\{a,c\},\{o\}}, M_{\{b,c\},\{p\}}), \underline{w}_{\{a,b\}}), \underline{P}(\underline{M_{\{a,b,c\},\{o,p\}}}, \underline{w}_{\{a,b\}})) = \mathit{false}$

7.2.4. Discussion

We will also refer to this use of partial mappings for (in)dependent subtasks as **the use of subsystems**.

Observe that the partial mappings each are responsible for part of the behavior of the total system and that the output value assignment of the system is due to both partial mappings. This is very similar (or even equal) to what the behavioral layers (input-output couplings) of a reactive system do.

Thus, we can draw the following conclusions. A reactive system, formalized by means of partial mappings for behavioral layers, is an effective way to reduce hardware requirements if (in)dependent subtasks are present. Our findings, interpreted as such, thus not only show that Reactive Robotics is an effective way to reduce resource requirements, but as well define a condition under which that is true (presence of subtasks).

7.3. Context

As could be seen in the previous section, dependent subtasks – i.e. subtasks with overlapping input sets – can be effectively handled by two partial mappings that both take the overlapping variables into account.

Now imagine the special case in which those overlapping variables could be used (and 'processed') by both of those partial mappings in the same way. In what follows we exemplify this special case that is commonly referred to as context. Furthermore, we discuss that there is some redundancy in both partial mappings

[8] If $\prod_{v^I \in V^{I1} \setminus V^{I\prime}} \lfloor |D_{v^I}| \rfloor = 2$ and $\prod_{v^I \in V^{I2} \setminus V^{I\prime}} \lfloor |D_{v^I}| \rfloor = 2$ (i.e. for the simplest case where $|V^{I1} \setminus V^{I\prime}| = |V^{I2} \setminus V^{I\prime}| = 1$ and all variables are binary (have a domain of size 2)) there will be no reduction (as $a * (2 * 2) = a * (2 + 2)$), but otherwise the strength of the reduction will increase strongly with the number of and size of the domain of the variables

taking those overlapping variables into account separately (the 'processing' thus being done twice). To reduce this redundancy we then propose to preprocess the overlapping variables and proof the effectiveness thereof.

Context for behavior. Context is not something that one can run into. Rather it is the role something plays to a system. Consider for example a museum. The museum *is* no context, but *plays the role of* context to a visitor when she is responding to an object (i.e. piece of art) in it. We will try to clarify this in the following example.

Example of context. Consider for example the mapping for a brute force system in Figure 7 in which $\{c,d\}$ acts as context to subtasks $\{a\}$,$\{o\}$ and $\{b\}$, $\{p\}$.

This response pattern can be represented as the following conditional statement-structure keyed on arbitrary Boolean formulas for input value assignment $\underline{w}_{\{a,b,c,d\}}$;
. if $(\underline{w}_{\{a,b,c,d\}}(c) = 0 \ xor \ \underline{w}_{\{a,b,c,d\}}(d) = 0)$ then
. if $\underline{w}_{\{a,b,c,d\}}(a) = 0$ then the produce call should assign 0 to o
. if $\underline{w}_{\{a,b,c,d\}}(a) = 1$ then the produce call should assign 1 to o
. if $\underline{w}_{\{a,b,c,d\}}(b) = 0$ then the produce call should assign 0 to p
. if $\underline{w}_{\{a,b,c,d\}}(b) = 1$ then the produce call should assign 1 to p
. if $\neg(\underline{w}_{\{a,b,c,d\}}(c) = 0 \ xor \ \underline{w}_{\{a,b,c,d\}}(d) = 0)$ then
. if $\underline{w}_{\{a,b,c,d\}}(a) = 0$ then the produce call should assign 1 to o
. if $\underline{w}_{\{a,b,c,d\}}(a) = 1$ then the produce call should assign 0 to o
. the produce call should assign 0 to p

So one could say that there are different states the context can be in (in this case $\underline{w}_{\{a,b,c,d\}}(c) = 0 \ xor \ \underline{w}_{\{a,b,c,d\}}(d) = 0$ and its negation), on which the response pattern of different subtasks is dependent in a consistent way (here the response pattern to a is inverted, whereas the response pattern to b is inhibited if $\underline{w}_{\{a,b,c,d\}}(c) = 0 \ xor \ \underline{w}_{\{a,b,c,d\}}(d) = 0$).

Observe that the example in Figure 6 that we used before, $\{c\}$ is context to subtasks $\{a\}$, $\{o\}$ and $\{b\}$, $\{p\}$.

7.3.1. Formal definitions

We define an input set $V^{I\prime}$ subset of V^I to be **context** to a mapping M'_{V^I,V^O} if there exists values (representing aforementioned states) for a variable i for each submapping in M'_{V^I,V^O} such that these can (1) substitute $V^{I\prime}$ and (2) be derived from $V^{I\prime}$. Here, substitutability is intended to mean that if we know the value for i, we can ignore the values for $V^{I\prime}$; $\forall_{m^1_{V^I,V^O}, m^2_{V^I,V^O} \in M'_{V^I,V^O}} [\underline{O}(m^1{}_{V^I,V^O}{}^I, \ m^2{}_{V^I,V^O}{}^I, \ \{i\} \cup V^I \setminus V^{I\prime}) \rightarrow \underline{O}(m^1{}_{V^I,V^O}{}^O, \ m^2{}_{V^I,V^O}{}^O, \ V^O)]$. Derivability is intended to mean that we can derive the value of i from the values for $V^{I\prime}$: $\forall_{m^1_{V^I,V^O}, m^2_{V^I,V^O} \in M'_{V^I,V^O}} [\underline{O}(m^1{}_{V^I,V^O}{}^I, \ m^2{}_{V^I,V^O}{}^I, \ V^{I\prime}) \rightarrow$ *value of* i *for* $m^1{}_{V^I,V^O} =$ *value of* i *for* $m^2{}_{V^I,V^O}]$

Notice that with this definition of context, one can treat every $V^{I\prime} \subseteq V^I$ as context to a mapping M'_{V^I,V^O}. The easiest way to do so, is to introduce for each combination of values for variables in $V^{I\prime}$ a value in D_i and use that value for each submapping in which the variables in $V^{I\prime}$ have that combination of values. Even thought the domain of i now is rather big, it can clearly substitute $V^{I\prime}$ and be derived from it. In other words, $V^{I\prime}$ is context.

However, as we will show later on, the introduction of an i is only useful (for reducing the number of submappings) if D_i is smaller than $\prod_{v^I \in V^{I\prime}} |D_{v^I}|$.

7.3.2. Example

In Figure 7 one can find the partial mappings used to handle a task with context by combining over context (under the headings with 'preprocessing').

7.3.3. Results

With the use of variable i we can combine over the context. This results in a produce call that first derives the value for i from the context via a nested produce call and then uses that value to derive the correct output value assignment (see next paragraph). Under certain constraints this results in a reduction of the number of submappings used (see the paragraph after that).

Combining over context. Assume a mapping M'_{V^I, V^O} that consists only of subtasks V^{I1}, V^{O1} and V^{I2}, V^{O2} overlapping in context $V^{I\prime}$, that can be substituted by a variable i that can be derived from it. By considering i part of the output set, we can combine over $V^{I\prime}$ and i due to the derivability constraint; $\underline{C_{over}}(M'_{V^I, V^O}, V^{I\prime}, \{i\})$. By considering i part of the input set, we can combine over all input variables (including i) except for the context and the output set V^O due to the substitutability constraint: $\underline{C_{over}}(M'_{V^I, V^O}, \{i\} \cup V^I \setminus V^{I\prime}, V^O)$.

Now we need to connect these such that i is used only internally, which can be done by the following produce call, which arranges nicely that indeed i is an input variable for the one and an output variable for the other mapping; $\underline{P}(\underline{X}(\underline{C_{over}}(M'_{V^I, V^O}, \{i\} \cup V^I \setminus V^{I\prime}, V^O), \{i\} \cup V^I \setminus V^{I\prime}, V^O), \underline{MA}(\underline{P}(\underline{X}(\underline{C_{over}}(M'_{V^I, V^O}, V^{I\prime}, \{i\}), V^{I\prime}, \{i\}), \underline{w}_{V^{I\prime}}), \underline{w}_{V^I \setminus V^{I\prime}}))$

Note the use of the unite value assignments function (MA) to unite the value for i produced by the nested produce call and the rest of the input value assignment. Note as well the use of the cut function (X) to crop the mappings used, which will allow for a clearer graphical representation of the used mappings later on.

Observe that we can now combine in the first mapping (the one that uses i as an input variable) over the subtasks, just like we did before. These subtasks will now only be dependent in $\{i\}$.

Reduction of size by combining over context. Consider the two mappings in the produce call above. The nested mapping (that produces a value for i) will in the worst case contain a submapping for each combination of values for the variables in the context $V^{I\prime}$ (note that this might be improved on); $|\underline{C_{over}}(M'_{V^I, V^O}, V^{I\prime}, \{i\})| = \prod_{v^I \in V^{I\prime}} \lceil |D_{v^I}| \rceil$. The other mapping (that uses the value for i) will in the worst case contain a submapping for each combination of values for i and the variables in the input set not in the context $V^I \setminus V^{I\prime}$ (note that this might be improved on, for example by combining over the subtasks); $|\underline{C_{over}}(M'_{V^I, V^O}, \{i\} \cup V^I \setminus V^{I\prime}, V^O)| = \prod_{v^I \in V^I \setminus V^{I\prime}} \lceil |D_{v^I}| \rceil * |D_i|$.

The sum of these is the total number of submappings used; $\prod_{v^I \in V^{I\prime}} \lceil |D_{v^I}| \rceil + \prod_{v^I \in V^I \setminus V^{I\prime}} \lceil |D_{v^I}| \rceil * |D_i|$.

We will now compare this to the number of submappings used by a brute force system ($\underline{P}(M'_{V^I, V^O}, \underline{w}_{[I]'})$) that is defined for all possible input value assignments to V^I ($= |Dom^I_{\underline{P}(M'_{V^I, V^O}, \underline{w}_{[I]'})}| = \overline{\prod_{v^I \in V^I} \lceil |D_{v^I}| \rceil}$), which can be (re)written as; $\prod_{v^I \in V^I \setminus V^{I\prime}} \lceil |D_{v^I}| \rceil * \prod_{v^I \in V^{I\prime}} \lceil |D_{v^I}| \rceil$. Now it is easy to observe that if the domain of i is sufficiently much smaller than is $\prod_{v^I \in V^{I\prime}} \lceil |D_{v^I}| \rceil$, this will yield a reduction of the number of submappings used. Note that this reduction becomes bigger as the domain of i becomes smaller.

What is more, each of the submappings in this new mapping only has to determine the value for part of the output set (on the basis of only part of the input set), which will presumably reduce the resource requirements even further.

Combining over context and subtasks. As mentioned before, apart from combining over context, we can combine over subtasks as well. For the mapping discussed above, this would result in the following produce call; $\underline{P}(\underline{\oplus}(\underline{X}(\underline{C_{over}}(M'_{V^I, V^O}, \{i\} \cup V^{I1} \setminus V^{I\prime}, V^{O1}), \{i\} \cup V^{I1} \setminus V^{I\prime}, V^{O1}), \underline{X}(\underline{C_{over}}(M'_{V^I, V^O}, \{i\} \cup V^{I2} \setminus V^{I\prime}, V^{O2}), \{i\} \cup V^{I2} \setminus V^{I\prime}, V^{O2})), \underline{MA}(\underline{P}(\underline{X}(\underline{C_{over}}(M'_{V^I, V^O}, V^{I\prime}, \{i\}), V^{I\prime}, \{i\}), \underline{w}_{V^{I\prime}}), \underline{w}_{V^I \setminus V^{I\prime}}))$.

The number of submappings required for this produce call can now be expressed as; $\prod_{v^I \in V^{I\prime}} \lceil |D_{v^I}| \rceil + (\prod_{v^I \in V^{I1} \setminus V^{I\prime}} \lceil |D_{v^I}| \rceil + \prod_{v^I \in V^{I2} \setminus V^{I\prime}} \lceil |D_{v^I}| \rceil) * |D_i|$.

Assuming all conditions for effective combining over subtasks and context hold, this is once more a big reduction of size, bigger than that of just combining over subtasks or just combining over context.

See Figure 7 for a more concrete example and comparison of the different ways to combine over context and subtasks as well as their effectiveness in reducing the number and complexity of submappings.

7.3.4. Discussion

We have found that if a set of variables can be substituted by a value for a variable i derived from it (i.e. is context) and the domain of that variable i is sufficiently small, combining over that set can once more yield a reduction of the number of submappings used. We will refer to the process of combining over that set as it has been described above as **preprocessing**. We believe that this use of the word preprocessing fits nicely with the common use of the term, as the nested mapping produces an output value (for i) that is then used by another mapping as an input value.

As the term preprocessing suggests, the method here used is very similar to the use of preprocessing in hierarchical systems[9]. Thus, we can draw the following conclusions. A hierarchical system, formalized as using preprocessing as above, is an effective way to reduce hardware requirements if contexts with sufficiently small domains can be found. Our findings, interpreted as such, thus not only show that Hierarchical Robotics is an effective way to reduce resource requirements, but as well define a condition under which that is true (presence of contexts).

When using both preprocessing and partial mappings, another interpretation becomes possible. As before, we can interpret the partial mappings for the subtasks as the behavioral layers of a reactive system. On top of that, we now have the preprocessing partial mapping that is used to adapt the output values produced by those behavioral layers (as those behavioral layers take i into account). In other words, the preprocessing regulates the behavioral layers. This is very much in line with the Regulated Reactive Approach. Our findings, interpreted as such, thus not only show that the Regulated Reactive Approach is an effective way to reduce resource requirements, but as well define the conditions under which that is true (presence of subtasks and/or contexts).

8. Relating the found ways of combining to robotics approaches

First of, the following observation; multiple ways of combining as discussed in the previous Section can be used in the same system. That is, one can ignore several irrelevant variables one after another, preprocess one context, separate between subtasks in one of the resulting partial mappings and then preprocess another context – provided these irrelevant variables, contexts and subtasks are present. Furthermore, each of this ways of combining can yield a further reduction in the number of submappings used. A more concrete example of this can be found in Figure 7, where a combination of preprocessing and the use of partial mappings uses less submappings than preprocessing or the use of partial mappings on their own.

With this observation in place, we can now recall the way in which the different approaches to robotics have been related to ways of combining (see also Figure 8 and 9).

- the use of partial mappings for subtasks was found to be very similar to the way in which the behavioral layers of Reactive Robotics,
- the use of preprocessing was found to be very similar to the pre- and post-processing of Hierarchical Robotics,
- and the use of both partial mappings and preprocessing was found to be very similar to the behavioral layers and regulation of Regulated Reactive Robotics.

We believe that by describing the approaches to robotics in terms of the ways of combining related to them in the list above, we capture their main characteristics. On the other hand, there probably are various characteristics of different architectures derived from the approaches to robotics not captured by such a description[10]. However, all these (derived) architectures will likely still use preprocessing (if Hierarchical Robotics) or subsystems (if Reactive Robotics). Thus, unless those other characteristics are somehow

[9] And the postprocessing of such hierarchical systems can easily be interpreted as preprocessing as well.

[10] One notable example is the subsumption architecture[3]. Because some mechanisms for combining the output value assignments of different behavioral layers are defined in that architecture, one could say that it contains some form of regulation/preprocessing – albeit limited and restricted.

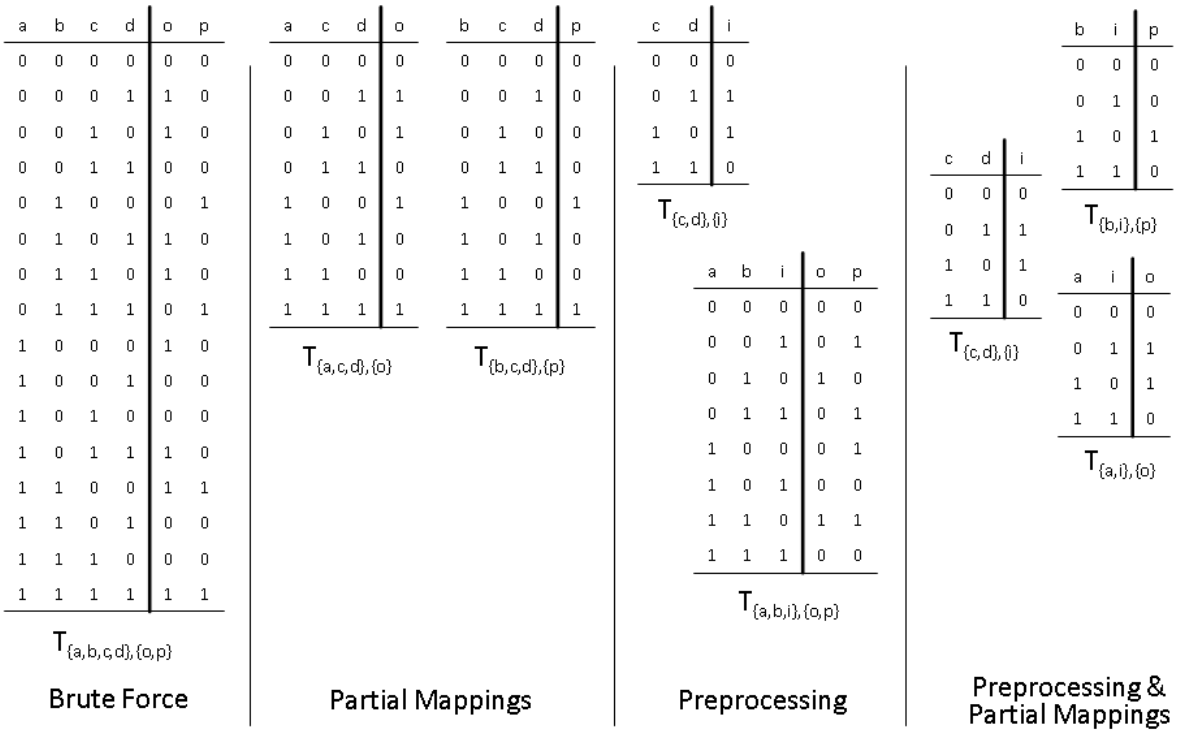
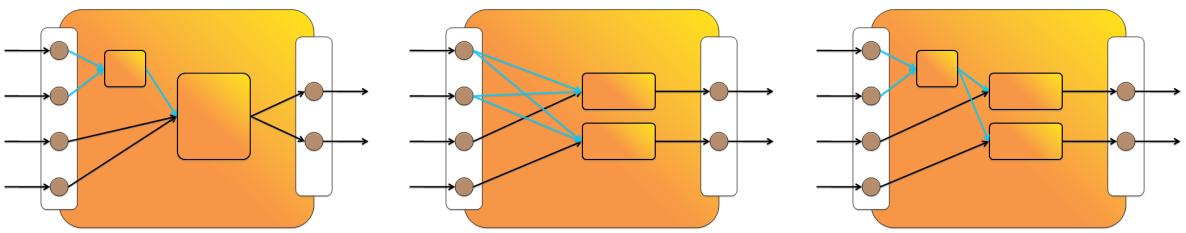


Figure 7: Four different mappings that (with the proper produce call) can all produce the same behavior. Here $\{a,c,d\}$, $\{o\}$ and $\{b,c,d\}$, $\{p\}$ are subtasks, with $\{c,d\}$ as context. The mapping to the left consists of only basic mappings and could thus be used by a brute force system. Combining over subtasks in that mapping would result in the partial mappings next to that. Combining over context would result in the partial mappings second to the right, of which the left one preprocesses the context. Combining over both subtasks and context would result in the partial mappings to the right, of which the left one preprocesses the context. Compared to the other (partial) mappings, these latter partial mappings use the least submappings in total. As is shown in the text, this difference will only get bigger as the input set gets larger (provided combining over subtasks and context is still possible).



(a) Hierarchical Robotics (b) Reactive Robotics (c) Regulated Reactive Robotics

Figure 8: Graphical representations of the way in which the different robotic approaches utilize the ways of combining. In each of the figures the input variables (circles to the left), output variables (circles to the right) and partial mappings used (boxes in between) and the way they are structured (arrows in between) are shown. The top two input variables (denoted by lighter arrows) can be effectively used as context variables. Note that the systems here displayed could well use the partial mappings defined in Figure 7.

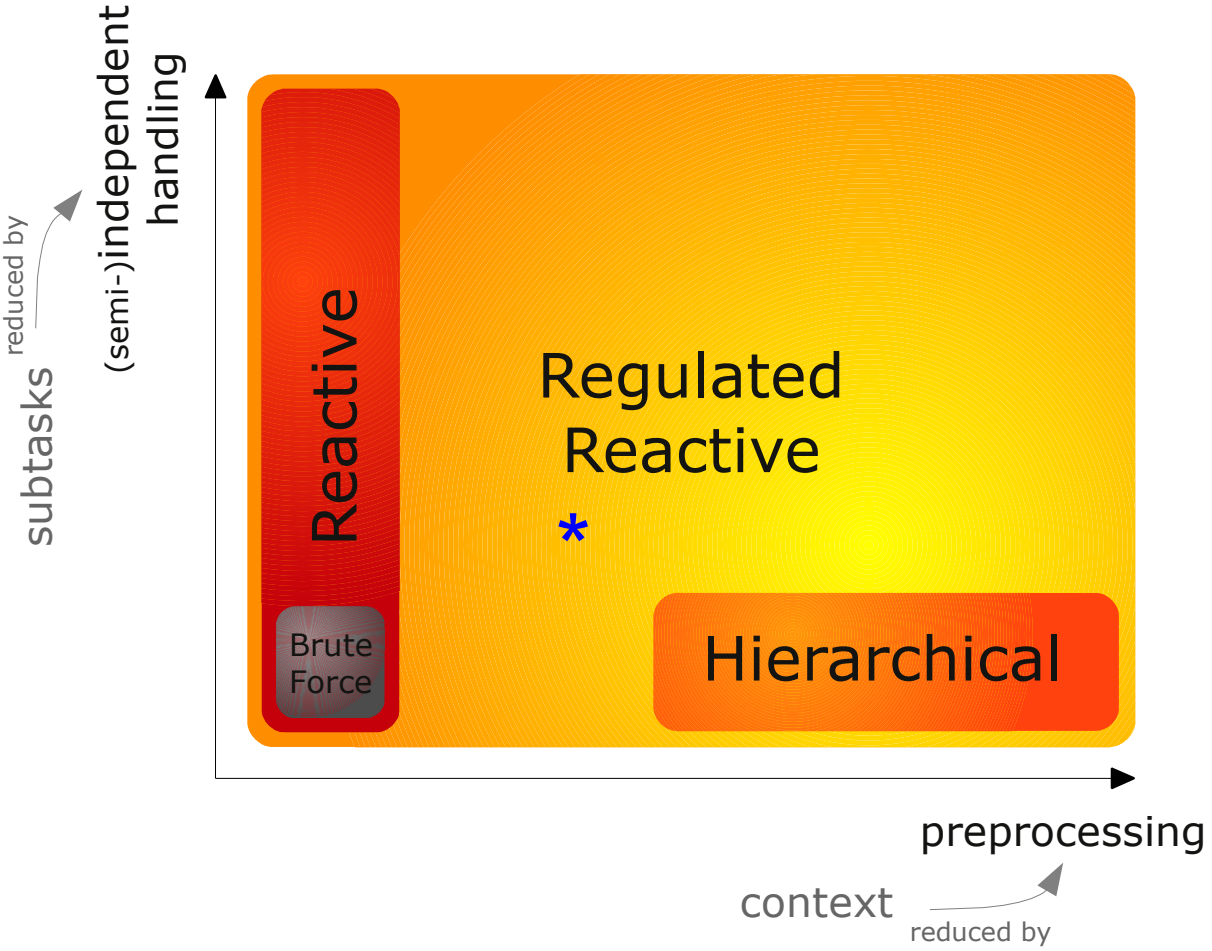


Figure 9: Here all systems that display a particular behavior can be plotted according to the extend in which they use either preprocessing (x-axis) and/or subsystems (y-axis). Note that all these systems would have the same behavior, yet different produce calls and thus different amounts of submappings used and thus different hardware requirements. Brute Force systems (bottom-left) are those that use subsystems nor preprocessing. Reactive systems (left) are those that could use subsystems but no preprocessing. Hierarchical systems (bottom) are those that use some preprocessing but no subsystems. Regulated Reactive systems (all) are those that could use subsystems as well as preprocessing. A regulated reactive system (such as the one denoted by $\star$) could well use less submappings than any reactive or hierarchical system, if both some subtasks and context are present.

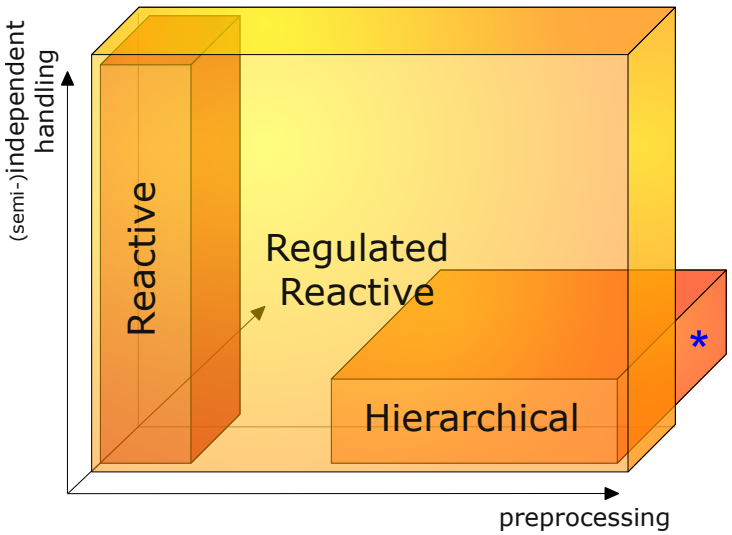


Figure 10: This Figure is the same as Figure 9, with the addition of a third axis. Imagine that one would find another way of combining over (represented by the third axis), that strongly reduces the number of submappings needed but somehow is not compatible with a (Regulated) Reactive approach - assuming that such an incompatibility is possible. Then it could be that there exists for particular tasks an hierarchical system (such as the one denoted by $\star$) that requires less submappings than does any reactive of regulated reactive system. Likewise, another way of combining over could be used to have a reactive system require less submappings than any hierarchical or regulated reactive system. Notice that this is only possible if such an incompatible way of combining exists.

conflicting (see Figure 10), what we say about the approaches to robotics described in terms of these ways of combining will also apply to those derived architectures. Therefore we believe it justified to in the remainder of this paper describe the approaches to robotics as such.

In combination with the observation made above, such a description of the approaches to robotics allows for clear-cut conclusions. For we have shown that the use of subsystems and preprocessing reduces hardware requirements. Thus, our findings imply that both Reactive and Hierarchical Robotics use effective ways to reduce hardware requirement and that Regulated Reactive Robotics reduces hardware requirements even more effective as it uses a combination of subsystems and preprocessing.

Furthermore, we can observe that Reactive and Hierarchical Robotics, when described as we did, are special cases of Regulated Reactive Robotics (with no preprocessing and no subsystems respectively). Consequently, Reactive and Hierarchical Robotics will never require less resources than does Regulated Reactive Robotics.

9. Discussion

In this paper we have briefly discussed Reactive and Hierarchical Robotics (introduction) and introduced the new Regulated Reactive Robotics (Section 1). Furthermore, we have introduced a new formalism for discussing the and comparing different systems on the structure level (Section 2 and 3). This formalism was used to show that all approaches to robotics here discussed have characteristics that result in a reduction of their hardware requirements (in comparison to a brute force system) (Section 4, 5, 6, and 7). These reductions were caused by using subsystems for subtasks (in reactive systems), using preprocessing for context (in hierarchical systems), or both (regulated reactive systems) (Section 8). Thus, in this respect, both Reactive and Hierarchical Robotics can be considered special cases of Regulated Reactive Robotics that will in particular cases (with subtasks and/or context) always require more hardware than does a system from Regulated Reactive Robotics.

These findings are relevant from a roboticists perspective for several reasons. First of we did not only introduce Regulated Reactive Robotics, but have shown its effectiveness in reducing hardware requirements as well. Furthermore, we found conditions under which subsystems and preprocessing can effectively be applied (subtasks and context respectively). Roboticists could well use these findings to decide which approach is best for the desired behavior. For example, a robot that is to take many contexts into account with few opportunities for distinguishing subtasks might, given our findings, be best served by an hierarchical or regulated reactive system. And last but not least, the formal framework could well be used to help find and prove the effectiveness (in terms of reducing hardware requirements) of other ways of combining.

Furthermore, there are exciting possibilities for extending the scope and applicability of this work. One such possibility is the investigation of systems that take time into account. This could be done by having preprocessing partial mappings take into account (as an input variable) the value they have produced before (for an output variable), i.e. by introducing loops. Another such possibility would be an investigation of the difficulties and possibilities of developing or learning the structure of a regulated reactive system. As has been shown using computational complexity theory that under many conditions doing so will be NP-hard [10] trying do do so might well pose interesting challenges. One notable example of learning in a structure that shows remarkable resemblance to Regulative Robotics (incorporating temporal structure as well) is the one by Yamashita and Tani [11].

To conclude, we would like to make some final remarks on the generalizability of our findings and framework. For even though we here interpreted those from a roboticist perspective, no assumptions were made that would require one to do so. Thus, our research can be extended to all systems, including, for example, computer programs, organizations and cognitive systems. And for all systems the conclusion would be valid that it probably is beneficial for the hardware requirements to use subsystems for subtasks and to preprocess context. In other words, it is probably beneficial to use a Regulated Reactive approach.

References

[1] N. Nilsson, C. Rosen, B. Raphael, G. Forsen, L. Chaitin, S. Wahlstrom, Application of Intelligent Automata to Reconnaissance, Technical Report, Stanford Research Institute, 1968.

[2] J. A. Fodor, Précis of the modularity of mind, The Behavioral and Brain Sciences 8 (1985) 1–42.

[3] R. A. Brooks, Elephants don't play chess, Robotics and Autonomous Systems 6 (1990) 3–15.

[4] P. Haselager, J. van Dijk, I. van Rooij, A lazy brain? embodied embedded cognition and cognitive neuroscience., in: P. Calvo, A. Gornila (Eds.), Handbook of Cognitive Science, an Embodied Approach, Elsevier Inc., 2008.

[5] J. van Dijk, R. Kerkhofs, I. van Rooij, P. Haselager, Can there be such a thing as embodied embedded cognitive neuroscience?, Theory and Psychology 18 (2008) 297–316.

[6] S. Lagarde, *Evolving a circadian rhythm.* Bachelor's thesis, Department of Artificial Intelligence, Radboud University Nijmegen, Nijmegen, 2009.

[7] L. Bax, *Cognitive control in reactive agents: Surviving predators through the evolution of a circadian rhythm.* Bachelor's thesis, Department of Artificial Intelligence, Radboud University Nijmegen, Nijmegen, 2010.

[8] D. Marr, Vision: A computational investigation into the human representation and processing of visual information., San Francisco: W.H. Freeman and Company., 1982.

[9] R. McClamrock, Marr's three levels: A re-evaluation, Minds and Machines 1 (1991) 185–196.

[10] T. Wareham, J. Kwisthout, P. Haselager, I. van Rooij, Ignorance is bliss: A complexity perspective on adapting reactive architectures, Accepted to the First IEEE Conference on Development and Learning and on Epiginetic Robotics, August 24-27, Frankfurt am Main, Germany. (2011).

[11] Y. Yamashita, J. Tani, Emergence of functional hierarchy in a multiple timescale neural network model: a humanoid robot experiment, PLoS Computational Biology 4 (2008) e1000220.