MASTER

Definition and simulation of supervisory control models in Haskell

Bernts, I.T.D.

Award date:
2020

Eindhoven University of Technology

Master Thesis

Definition and simulation of supervisory control models in Haskell

Author:
Ivo Bernts

Supervisors:
Dr. T. Verhoeff
Dr. J.M. van de Mortel-Fronczak

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in

Computer Science and Engineering

August 17, 2020

Abstract

CIF3 is a domain specific language for specifying system controllers developed and maintained by the Control System Technology group at the Department of Mechanical Engineering. Specifications are defined as instances of discrete event systems. In the CIF3 language, discrete event systems are described in the form of automata. The current implementation of the CIF3 toolchain, which is written in Java within the Eclipse Modeling Framework, has some maintainability and extendibility issues. The goal of this master project is to discover how a functional programming language can be used to build an alternative for CIF3. We do this by implementing a proof of concept of our own DSL, named X-Control, with an accompanying toolchain in Haskell.

Before we start on the design of X-Control, we discuss its theoretical background. We discuss algebraic automata theory, which is an approach to automata theory proposed by Samuel Eilenberg. We also discuss an extension of algebraic automata theory called X-machines (which is also introduced by Samuel Eilenberg). In X-machines, the labels on the transition edges of automata correspond with binary relations on some arbitrary domain. This means behavior can be partly modeled in how the relations 'manipulate' domain values.

We also discuss the existing theory on which CIF3 is based. We discuss the definition of discrete event systems, and how they are used to model a *plant* (the total physical behavior of the system) and the *requirements* (the required behavior). We also discuss the concept of supervisory control synthesis, which is the process of generating a controller (supervisor) for the system that makes sure that the to-be-controlled system (modeled by the plant) adheres to the requirements. We discuss two existing formalisms which are used to model discrete event systems. The first of which is the Finite State Automaton (FSA) formalism, which is an elementary automaton-based formalism. The labels of the automata are interpreted as events (which correspond to possible interactions of the system with its environment). The second discussed formalism, is the Extended Finite Automaton (EFA) formalism. This is also the formalism on which the syntax of CIF3 is based. In EFAs, the transition edges not only contain a label (event), but also a guard and an update function on some arbitrary domain. In practice this domain consists of a number of variables (which are comparable to variables in programming). For both FSAs and EFAs we discuss a synchronous product operator, and a supervisory control synthesis algorithm. The synchronous product operator can be used to compose multiple systems. This allows one to break up a complex system into multiple subcomponents. For EFAs we also discuss some current limitations, which also occur in the CIF3 language.

We then introduce our own formalism, based on X-machines, which we call D-systems. D-systems will be the underlying formalism for our own DSL X-Control. In D-systems events are not modeled as labels, but rather as binary relations on an arbitrary domain. A D-system then consists of sets of controllable and uncontrollable events (both sets of binary relations on D), a set of initial domain values and a D-EventMachine (which is a slight alteration of X-machine). As done with FSAs and EFAs, we define a synchronous product operator for D-systems. We discuss how one can use D-systems do model a plant and the requirements for some system, while addressing the limitations of the EFA formalism. We also introduce an operator for restricting the behavior for D-systems. This operator is useful since separate subcomponents of a plant can restrict each

other in physical situations. Lastly, we introduce a supervisory control synthesis algorithm for D-systems.

After discussing our formalism, we discuss the design and implementation of our language. First we discuss the CIF3 language and toolchain in more detail. We then discuss the approach for designing and implementing a prototype for X-Control. We choose to follow a semantics-driven approach, which means we first implement our semantic domain, after which we create syntax for all elements of our semantic domain. The subdomains (types) and operations in our semantic domain correspond with the definitions introduced in our D-system formalism. We create an *internal syntax* for our language, which means that the language exists within the host language (Haskell). We implement a small toolchain, containing a simulator and the implementation of the supervisory control synthesis algorithm. Lastly, we propose a number of possible extensions for X-Control, while giving suggestions how these extensions could be implemented.

Acknowledgments

I would like to thank my supervisors Tom Verhoeff, Asia van de Mortel-Fronczak and Koos Rooda for all the guidance, feedback and support during this master project. I would also like to thank Ferdie Reijnen and Martijn Goorden for the feedback and advice, and Albert Hofkamp for helping me get started with CIF3.

In this project, the following tools where used.

- LaTeX, for typesetting,
- The Glasgow Haskell Compiler, for debugging Haskell code,
- IHaskell[1], which is a kernel for the Jupyter project, which allows one to use Haskell in a Jupyter Notebook. IHaskell is used during the development of the semantic domain, and the syntax of X-Control. From the resulting notebooks, Appendix C, and Appendix D are generated,
- Ipe and Tikz for image creation.

[1]The project page of IHaskell can be found at `https://github.com/gibiansky/IHaskell`.

Contents

Part I

Preamble

Chapter 1

Introduction

1.1 Context

This Computer Science master project was carried out at the Department of Mechanical Engineering, in the Control Systems Technology (CST) group. At the CST group, the topic of system control is studied, which concerns the development of control software. Their project consists of, among other things, the control software for waterway locks [20] [21].

For the development of control software (also called ‘controllers’), there are two options. One is to build the controller by hand, and the other is to generate the controller from a specification automatically. The latter is preferred since it is less error prone. This specification consists of a *plant*, which specifies the possible physical behavior of the system, and the *requirements*, which specify the required behavior of the system. Given the plant and the requirements, a controller (also called a *supervisor*) which makes sure that the system adheres to the requirements can then be generated. This process is called *supervisory control synthesis*. Both the plant and the requirements are specified in the form of a *discrete event system* (DES).

A discrete event system is a discrete-state, event-driven system which is often modeled as an instance of (an extension of) *finite automata*. The events are depicted as labels on the transition edges of the automaton. Each event corresponds with some interaction of the system with its environment. An event can then either be controllable or uncontrollable. Controllable events are initiated by the system (e.g. switching a motor or light source on or off), implying that the system has control over these events. Uncontrollable events are initiated by the environment (e.g. some button is pressed or a sensor value reaches some threshold), implying that the system has no control over these events.

The CIF3 language created by the CST group allows end users to specify the plant and the requirements in the form of automata extended with variables, transition guards and transition update functions. The CIF3 tooling can then be used to generate a supervisor using supervisory control synthesis. Plants and generated supervisors can then be validated through simulation. CIF3 and its underlying theory will be our main points of attention during this project. In Section 1.3 we will further introduce the CIF3 language and tooling.

1.2 Domain Specific Languages

Programming languages can be *domain specific* instead of general purpose, as discussed in [15]. These domain specific languages (DSLs) (of which CIF3 is an example), are specialized in a certain domain. This specialization is done by trading generality for expressiveness in this limited domain. This expressiveness is achieved by introducing notations and syntax constructs specifically tailored to the domain. This also greatly increases the ease of use compared to the general purpose languages for the specific domain, which leads to increased productivity and reduced maintenance costs. Well known examples of DSLs are

- HTML, which is a language for creating hypertext web pages,
- LaTeX, which is a typesetting language,
- Make, which is language for specifying how some piece has to be built from its source code,
- SQL, which is a language for defining relational database queries.

For the sake of comparison, two well known examples of general purpose languages are C++ and Java.

1.3 The CIF Project

The Compositional Interchange Format 3 (CIF3) is a domain specific language for defining (among other things) *Discrete Event Systems*. Since these systems are modeled as automata, the CIF3 language is mostly automata-based. The events (the transition edge symbols of the automata) represent the possible interactions the system can have with its environment. In CIF3, transition edges have guards and update functions on user-defined variables.

CIF3 comes with a toolchain written in Java. This toolchain is built within the Eclipse Modeling Framework (EMF), which is used for creating metamodels (in this case the metamodel would define the CIF3 language) in a graphical manner. The toolchain comes with an editing environment for creating and modifying models, a (graphical) simulator, validation tools, a supervisory control synthesis algorithm, and a number of code generation tools.

At the moment of writing, there is a number of issues regarding the toolchain, mainly concerning maintainability and extendiblity (which we will discuss in Chapter 7). To address these issues, we will consider an alternative for Java and the EMF framework.

1.4 Functional Programming Languages

For our alternative approach for designing and implementing a language for modeling discrete event systems, it is interesting to consider a functional programming language, since domain specific languages can be modeled in a compact way using the typing systems offered by functional languages. The monad design pattern can be used for keeping track of state information when performing simulations. This can, for instance, be done with the `State` monad. Because of these features, functional languages are particularly suitable to be used as *host languages* for DSLs, as shown by the following examples.

- Lava, a DSL implemented in Haskell, which is discussed in [6]. Lava is a DSL for specifying and designing circuits. The tool assists in verifying and implementing hardware.
- CλaSH, a DSL implemented in Haskell, which is discussed in [7]. CλaSH is also used for defining circuits. The CλaSH tooling provides a tool for synthesizing VHDL (a hardware descriptor language). For this tool, the API of the Glasgow Haskell Compiler is used to simplify descriptions created in CλaSH, which in turn simplifies the synthesizing process.
- FSMLanguage, a DSL discussed in [4] which is implemented in Haskell. FSMLanguage is a DSL used for hardware/software co-design for FPGAs.

- ExaSlang, a DSL implemented in Scala (a language with both object-oriented and functional features), which is discussed in [23]. ExaSlang is a DSL for defining solvers for High-Performance Computing systems (which are systems with multiple CPUs and complex memory architectures and accelerators).
- Harpy, a DSL within Haskell for generating x86 machine code at run-time, discussed in [12].
- An implementation of the language Orc in Haskell is discussed in [8]. Orc is a DSL specialized in the implementation of concurrent programs.

A disadvantage of functional programming languages is their steeper learning curve, which may make it more difficult to train future maintainers.

Chapter 2

Research Plan

2.1 Research Question

For this research project we will determine how a functional programming language can be used to implement a modeling language which will serve as an alternative for CIF. That is, how we can use a functional programming language to implement a DSL for modeling *discrete event systems*, which can then be simulated. Our research question for this project, which we will keep more general, is defined as follows.

- How can a functional programming language be used when developing tools for defining and simulating operational models, with maintainability and extendibility taken into consideration?

2.2 Approach

Before we look at actually implementing a DSL, we must first discuss the underlying mathematical formalism. This we will do in Part II of this report. Since discrete event systems are usually defined using automaton-like formalisms, we will discuss *Algebraic Automata Theory* proposed by Samuel Eilenberg in [10]. This approach to automata theory might be more suitable (than the traditional approach) for implementation in a functional programming language, because its algebraic style resembles a more functional approach. We will also discuss the *X-machine* formalism, which is an extension of algebraic automata theory where automata can do some computations on some arbitrary domain. X-machines are used for a number of different purposes. In [5] X-machines are used to model agent-based systems. In [14] the use of a variant of X-machines where different instances can communicate, for formal and modular specification of large systems is discussed. In [13] a test generation technique for systems which are specified with X-machines is discussed. An algorithm for simulating X-machines in a functional style is discussed in [19]. In Part II we will also discuss the definition of *Discrete Event Systems*, and two existing formalism for defining such systems (one of which forms the basis of the CIF3 language). Lastly, we will define our own formalism for defining discrete event systems based on X-machines.

In Part III we discuss the language and tooling for our DSL for defining discrete event systems. We will first briefly discuss the existing language and tooling of CIF3, particularly how the language relates to its underlying formalism (as discussed in Part II), and the current issues of the language and the toolchain. We will also discuss the design and implementation of our own language X-Control, which has our formalism based on X-machines (as introduced in Part II), as its basis.

Finally, in Part IV we will give an answer to our research question based on the result obtained in Part III. We will also discuss possible further work.

Part II

Theory

Chapter 3

Algebraic Automata Theory

In this chapter we will discuss *algebraic automata theory*, which is a different approach (than the traditional one) to automata theory. This approach was first proposed by Samuel Eilenberg in [10, pp. 12-24, 30-32].

3.1 Basic Definitions

We now discuss the definition of an automaton as introduced by Samuel Eilenberg.

Definition 3.1.1 Suppose we have set Σ. A Σ-*automaton* is defined as a quadruple (Q, I, T, δ), where $I, T \subseteq Q$ and δ is a relation with $\delta : Q \times \Sigma \to Q$.

The set Σ is called the *alphabet* of the automaton, and Q the set of *states* of the automaton where its subsets I and T are called the set of *initial* and *terminal* states respectively. δ is called the *transition relation* of the automaton. Suppose we have $q' \in \delta(q, \sigma)$ where $q, q' \in Q$ and $\sigma \in \Sigma$, then we say there is a *transition* from q to q' with *label* σ. A transition is often denoted as $q \xrightarrow{\sigma} q'$. We do not name δ, which means instead of writing $q' \in \delta(q, \sigma)$ we write $q' \in q\sigma$. If $|q\sigma| = 1$ then we can also write $q\sigma = q'$.

End of Definition

Example 3.1.1 Suppose we have $\{a, b, c\}$-automaton $A = (\{q_0, q_1, q_2\}, \{q_0\}, \{q_2\}, \delta)$, where δ is defined as

$$q_0 a = q_1$$
$$q_1 b = q_2$$
$$q_2 c = q_1$$

A visual representation of A can be found in Figure 3.1.1.

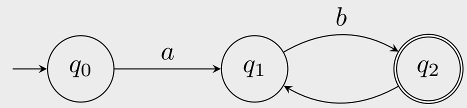

Figure 3.1.1: Visual representation of Example 3.1.1.

Example 3.1.2 Suppose we have $\{a, b, c\}$-automaton

$$A = (\{ q_0, q_1, q_2, q_3, q_4 \}, \{ q_0, q_1 \}, \{ q_0, q_3, q_4 \}, \delta)$$

δ is defined as

$$\begin{aligned} q_0 a &= \{ q_1, q_2 \} \\ q_1 b &= q_3 \\ q_2 c &= q_4 \\ q_4 b &= q_1 \end{aligned}$$

Again, a visual representation of A can be found in Figure 3.1.2.

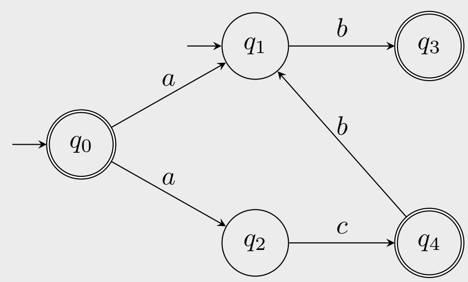

Figure 3.1.2: Visual representation of Example 3.1.2.

Example 3.1.3 Suppose we have $\{ a, b \}$-automaton $A = (\{ q_0, q_1, q_2 \}, \{ q_0 \}, \{ q_0, q_1 \}, \delta)$. δ is defined as

$$\begin{aligned} q_0 a &= q_1 \\ q_0 b &= q_2 \\ q_1 a &= q_2 \\ q_1 b &= q_0 \\ q_2 a &= q_2 \\ q_2 b &= q_2 \end{aligned}$$

Again, a visual representation of A can be found in Figure 3.1.3.

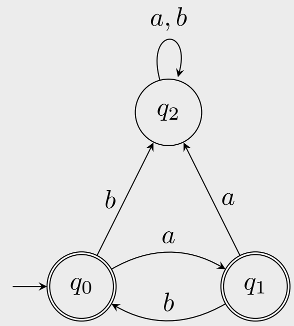

Figure 3.1.3: Visual representation of Example 3.1.3.

3.2 Behavior

A *path* (or *trace*) in some Σ-automaton A, is a sequence of transitions denoted by $p : q_0 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} q_n$, where for each i with $1 \leq i \leq n$ we have $q_{i-1}\sigma_i = q_i$. Just like a transition has a label $\sigma \in \Sigma$, a path also has a label. This label is the sequence of labels occurring in the path. A path label is then represented by an element of the *free monoid with base* Σ, which is defined as follows.

Definition 3.2.1 A *free monoid* with base Σ is defined as the monoid $(\Sigma^*, \cdot, \varepsilon)$, where $\cdot$ is not named in expressions. Σ^* is the set of n-tuples of elements of Σ (meaning that the value for n may differ for each element) which we write as $\omega = \sigma_1 \dots \sigma_n$ (with $n \geq 0$). Suppose $\omega = \sigma_1 \dots \sigma_n$ and $\omega' = \sigma'_1 \dots \sigma'_m$ then the product $(\cdot)$ $\omega\omega'$ is defined as

$$\omega\omega' = \sigma_1 \dots \sigma_n \sigma'_1 \dots \sigma'_m$$

The identity element ε is defined as the empty tuple.

End of Definition

The label $|p|$ of the path p, which is an element of Σ^*, is then denoted as

$$|p| = \sigma_1 \dots \sigma_n$$

p is *successful* if and only if $q_0 \in I$ and $q_n \in T$. Based on the concept of successful paths, the *behavior* of some automaton is defined as follows.

Definition 3.2.2 Suppose we have an Σ-automaton $A = (Q, I, T, \delta)$. The *behavior* of A, denoted as $\mathcal{L}(A)$, is a subset of Σ^*. For each $\sigma_1 \dots \sigma_n \in \mathcal{L}(A)$ there is a path (a sequence of transitions)

$$q_0 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} q_n$$

where, for i with $0 < i \leq n$, we have $q_{i-1}\sigma_i \ni q_i$ (meaning the transition $q_{i-1} \xrightarrow{\sigma_i} q_i$ exists), $q_0 \in I$, and $q_n \in T$.

End of Definition

Example 3.2.1 Suppose we have automaton A as given in Example 3.1.1. One can observe that the elements of the behavior $\mathcal{L}(A)$ are $ab, abcb, abcbcb, \dots$. We can express this behavior with a so called *regular expression*[a]:

$$\mathcal{L}(A) = ab(cb)^*$$

In this case the regular expression can be read as "each label must start with ab, after which 0 or more sequential repetitions of cb may occur".

[a] A quick introduction on regular expressions can be found on wikipedia: `https://en.wikipedia.org/wiki/Regular_expression`

Example 3.2.2 Suppose we have automaton A as given in Example 3.1.2. One can observe that we have the following for the behavior $\mathcal{L}(A)$.

$$\mathcal{L}(A) = \{\, \varepsilon, b, ab, ac, acbb \,\}$$

Example 3.2.3 Suppose we have automaton A as given in Example 3.1.3. One can observe that the elements of the behavior $\mathcal{L}(A)$ are $\varepsilon, a, ab, aba, abab, \ldots$. Again, we can express this behavior with a regular expression:

$$\mathcal{L}(A) = (ab)^*(a + \varepsilon)$$

In this case the regular expression can be read as "each label may start with 0 or more sequential repetitions of ab, after which a may (or may not) occur.

3.2.1 Operations

In this section the extension of the transition function δ is discussed. The resulting operation takes a state q and an element ω of Σ^*. The result of the operation is then a (set of) state(s) which are reached from q via a path with label ω. First a variant of this extension only applicable for *deterministic automata* (which is a more trivial case) is discussed. After which a variant applicable for all automata is considered.

A Σ-automaton (Q, I, T, δ) is said to be fully deterministic when $|I| = 1$ and δ is a *function* ($\forall q \in Q, \sigma \in \Sigma : |q\sigma| = 1$). An example of such a deterministic automaton is given in Example 3.1.3.

For deterministic automata, δ can be extended to obtain the following function θ.

Definition 3.2.3 Suppose we have a (deterministic) Σ-automaton $A = (Q, \{\, i \,\}, T, \delta)$, where δ is a function. We can then extend δ to obtain the function $\theta : Q \times \Sigma^* \to Q$ with

$$\begin{aligned} \theta(q, \varepsilon) &= q \\ \theta(q, \omega\sigma) &= \delta(\theta(q, \omega), \sigma) \end{aligned}$$

End of Definition

Just as with the transition relation δ, we do not name θ, meaning that instead of writing $\theta(q, \omega) = q'$ we write $q\omega = q'$. We can now express the behavior of some deterministic Σ-automaton $A = (Q, \{\, i \,\}, T, \delta)$ as

$$\mathcal{L}(A) = \{\, \omega \mid \omega \in \Sigma^*, i\omega \in T \,\}$$

Example 3.2.4 Suppose we have automaton A as given in Example 3.1.3. Observe that A is a deterministic automaton. We can now algebraically derive that, for example, aba is in $\mathcal{L}(A)$:

$$\begin{aligned} q_0aba &= (q_0ab)a \\ &= ((q_0a)b)a \\ &= (((q_0\varepsilon)a)b)a \\ &= ((q_0a)b)a \\ &= (q_1b)a \\ &= q_0a \\ &= q_1 \end{aligned}$$

Since q_0 is the initial state and q_1 is a terminal state we now conclude that $aba \in \mathcal{L}(A)$. Conversely, we can derive that abb is not in $\mathcal{L}(A)$:

$$\begin{aligned} q_0abb &= (q_0ab)b \\ &= ((q_0a)b)b \\ &= (((q_0\varepsilon)a)b)b \\ &= ((q_0a)b)b \\ &= (q_1b)b \\ &= q_0b \\ &= q_2 \end{aligned}$$

Since q_2 is not a terminal state, we can conclude that $abb \notin \mathcal{L}(A)$.

For any (non-deterministic) automaton we can extend the transition relation to obtain γ, which is defined as follows.

Definition 3.2.4 Suppose we have Σ-automaton $A = (Q, I, T, \delta)$. We can extend δ to obtain partial function $\gamma : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ with

$$\begin{aligned} \gamma(X, \varepsilon) &= X \\ \gamma(\emptyset, \omega) &= \emptyset \\ \gamma(\{\, q \,\} \cup X, \sigma) &= \delta(q\sigma) \cup \gamma(X, \sigma) \\ \gamma(X, \omega\sigma) &= \gamma(\gamma(X, \omega), \sigma) \end{aligned}$$

End of Definition

Again, we will not name γ, meaning that instead of writing $\gamma(X, \omega)$ we write $X\omega$. We can now express the behavior of some (non-deterministic) Σ-automaton $A = (Q, I, T, \delta)$ as

$$\mathcal{L}(A) = \{\, \omega \mid \omega \in \Sigma^*, I\omega \cap T \neq \emptyset \,\}$$

Example 3.2.5 Suppose we have automaton A as given in Example 3.1.2. We can now algebraically derive that, for example, ac is in $\mathcal{L}(A)$:

$$\begin{aligned}
\{\, q_0, q_1 \,\}\, ac &= (\{\, q_0, q_1 \,\}\, a)c \\
&= ((q_0 a) \cup (\{\, q_1 \,\}\, a))c \\
&= (\{\, q_1, q_2 \,\} \cup (\{\, q_1 \,\}\, a))c \\
&= (\{\, q_1, q_2 \,\} \cup ((q_1 a) \cup (\emptyset a)))c \\
&= (\{\, q_1, q_2 \,\} \cup (\emptyset \cup \emptyset))c \\
&= \{\, q_1, q_2 \,\}\, c \\
&= (q_1 c) \cup (\{\, q_2 \,\}\, c) \\
&= \emptyset \cup (\{\, q_2 \,\}\, c) \\
&= \{\, q_2 \,\}\, c \\
&= (q_2 c) \cup (\emptyset c) \\
&= \{\, q_4 \,\} \cup \emptyset \\
&= \{\, q_4 \,\}
\end{aligned}$$

Since $\{\, q_0, q_1 \,\}$ are the initial states and $\{\, q_4 \,\} \cap \{\, q_0, q_3, q_4 \,\} \neq \emptyset$, we conclude that $ac \in \mathcal{L}(A)$.

3.3 Relation to Functional Programming

Those who are familiar with list catamorphisms, as explained in [25, pp. 32- 42], can observe from the definition of θ that it can be defined as a *snoc list catamorphism*. We can interpret any element of Σ^* as a snoc list of elements of Σ ($\reflectbox{$\mathbb{L}$}.\Sigma$), where ε corresponds with the empty list $[\,]$ and $\omega\sigma$ corresponds with $\reflectbox{$\mathbb{L}$}_\omega \dashv \sigma$, where $\reflectbox{$\mathbb{L}$}_\omega$ is the snoc list interpretation of ω. Suppose we have a deterministic Σ-automaton (Q, I, T, δ) and some starting state q_0, then we have the following catamorphism on $\reflectbox{$\mathbb{L}$}.\Sigma$:

$$\begin{tikzcd}
\reflectbox{$\mathbb{L}$}.\Sigma \arrow[d, "f = (\!|\, q_0^\bullet \,\triangledown\, \delta \,|\!)"'] & & \mathbb{1} + \reflectbox{$\mathbb{L}$}.\Sigma \times \Sigma \arrow[ll, "{[\,]^\bullet \,\triangledown\, \dashv}"'] \arrow[d, "\mathsf{id}_{\mathbb{1}} + f \times \mathsf{id}_\Sigma"] \\
Q & & \mathbb{1} + Q \times \Sigma \arrow[ll, "q_0^\bullet \,\triangledown\, \delta"]
\end{tikzcd}$$

Observe that $\theta(q, \omega) = (\!|\, q^\bullet \,\triangledown\, \delta \,|\!).\reflectbox{$\mathbb{L}$}_\omega$.

If the transition relation δ is not a function, then we can interpret δ as a function with type $Q \times \Sigma \to \mathbb{L}.Q$. Knowing that $\mathbb{L}$ is a monad with *return function* $\eta : A \to \mathbb{L}.A$ and *bind function* $\lhd : (A \to \mathbb{L}.A) \to (\mathbb{L}.A \to \mathbb{L}.A)$, we can define the following function $g : \mathbb{L}.Q \to \Sigma \to \mathbb{L}.Q$

$$g.l.\sigma = \lhd\, .(\lambda q.\delta(q, \sigma)).l$$

Given a list of states l and a symbol σ, g will compute a list of all q' for which $q \xrightarrow{\sigma} q'$ where $q \in l$. For any (non-deterministic) automaton and some starting state set $Q_0 : \mathbb{L}.Q$, we have the following catamorphism:

$$
\begin{tikzcd}
\reflectbox{$\mathbb{L}$}.\Sigma \arrow[d, "f = (\!|\, Q_0{}^{\bullet} \triangledown g \,|\!)"'] & \mathbb{1} + \reflectbox{$\mathbb{L}$}.\Sigma \times \Sigma \arrow[l, "{[\,]}^{\bullet} \triangledown \dashv"'] \arrow[d, "\mathsf{id}_{\mathbb{1}} + f \times \mathsf{id}_{\Sigma}"] \\
\mathbb{L}.Q & \mathbb{1} + \mathbb{L}.Q \times \Sigma \arrow[l, "Q_0{}^{\bullet} \triangledown g"]
\end{tikzcd}
$$

Observe that $\gamma(X, \omega) = (\!|\, X^{\bullet} \triangledown g \,|\!).\reflectbox{$\mathbb{L}$}_{\omega}$.

The fact that θ and γ can be defined as catamorphisms, gives an indication that algebraic automata theory is suitable for functional programming languages.

Chapter 4

X-Machines

In this chapter we will discuss *X-machines*, which is a computational machine model proposed by Samuel Eilenberg in [10, pp. 266-272]. The X-machine formalism is an extension of algebraic automata theory as discussed in the previous section, where the edge labels correspond to relations on some arbitrary domain. For this reason, the X-machine could be considered as a computational model. We can use this computational model as a basis for our formalism for discrete event systems.

4.1 Basic Definitions

An X-machine consists of three components: the so called kernel, an input relation and an output relation. We first discuss the definition of the X-machine's kernel.

Definition 4.1.1 Suppose we have some arbitrary set X. An X-*machine kernel* is defined as a 5 tuple (Q, I, T, Φ, δ), where $I, T \subseteq Q$, $\Phi \subseteq \mathcal{P}(X^2)$ and δ is a relation of type $Q \times \Phi \to Q$.

End of Definition

Each $\phi \in \Phi$ is a *binary relation* on X. Suppose we have $x_1, x_2 \in X$ for which $(x_1, x_2) \in \phi$. We then say x_1 is related to x_2 in ϕ. Such a pair can also be denoted by $x\phi y$. The set Φ is called the *type* of the X-machine.

An X-machine kernel $M = (Q, I, T, \Phi, \delta)$ can be interpreted as Φ-automaton $M' = (Q, I, T, \delta)$. This means that all definitions, interpretations and operations on automata are also applicable for X-machines.

Example 4.1.1 Suppose we have free monoid $(\{a, b, c\}^*, \cdot, \varepsilon)$ with base $\{a, b, c\}$. Suppose we have $\{a, b, c\}^*$-machine kernel $M = (\{q_0, q_1, q_2\}, \{q_0\}, \{q_2\}, \Phi, \delta)$. The type of M is defined as $\Phi = \{\phi_a, \phi_b, \phi_c\}$, where ϕ_σ is defined as

$$\phi_\sigma = \left\{ (\sigma\omega, \omega) \mid \omega \in \{a, b, c\}^* \right\}$$

for each $\sigma \in \{a, b, c\}$. In essence ϕ_σ removes the first symbol from some label, in the case this first symbol is a σ. δ is defined as

$$\begin{aligned} q_0\phi_a &= q_1 \\ q_1\phi_b &= q_2 \\ q_2\phi_c &= q_1 \end{aligned}$$

A visual representation of M can be found in Figure 4.1.1. The behavior of M is

$$\mathcal{L}(M) = \phi_a\phi_b(\phi_c\phi_b)^*$$

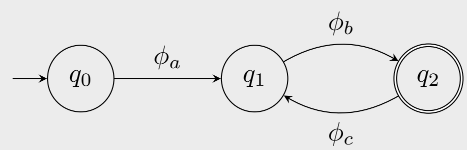

Figure 4.1.1: Visual representation of Example 4.1.1.

Example 4.1.2 Suppose we have $(\mathbb{Z}, \mathbb{Z})$-machine kernel $M = (\{ q_0, q_1, q_2 \}, \{ q_0 \}, \{ q_2 \}, \Phi, \delta)$. The type of M is defined as

$$\Phi = \{ \phi_*, \phi_-, \phi_0 \}$$

The definitions of the relations are as follows

$$\begin{aligned} \phi_* &= \{ ((n, m), (n, n * m)) \mid n, m \in \mathbb{Z} \} \\ \phi_- &= \{ ((n, m), (n - 1, m)) \mid n, m \in \mathbb{Z} \} \\ \phi_0 &= \{ ((0, m), (0, m)) \mid m \in \mathbb{Z} \} \end{aligned}$$

The relations can be interpreted as follows

- ϕ_*: Compute the product of the two integer values, and store the result in the second 'register'.
- ϕ_-: Decrement the first integer value.
- ϕ_0: Check whether the first integer value is equal to 0.

δ is defined as follows:

$$\begin{aligned} q_0\phi_* &= q_1 \\ q_0\phi_0 &= q_2 \\ q_1\phi_- &= q_0 \end{aligned}$$

A visual representation can be found in Figure 4.1.2. The behavior of M represented as regular expression is

$$\mathcal{L}(M) = (\phi_*\phi_-)^*\phi_0$$

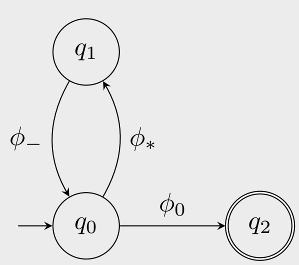

Figure 4.1.2: Visual representation of Example 4.1.2.

Example 4.1.3 Suppose we have free monoid $(\{a, b\}^*, \cdot, \varepsilon)$ with base $\{a, b\}$. Suppose we have $(\{a, b\}^* \times \mathbb{Z})$-machine kernel $M = (\{q_0, q_>, q_<\}, \{q_0\}, \{q_0\}, \Phi, \delta)$. The type of M is defined as

$$\Phi = \{\phi_+, \phi_-, (\phi_+;\phi_0), (\phi_-;\phi_0), (\phi_+;\phi_>), (\phi_-;\phi_<)\}$$

The definitions of the relations are as follows

$$\begin{aligned}
\phi_0 &= \left\{ ((\omega, 0), (\omega, 0)) \mid \omega \in \{a, b\}^* \right\} \\
\phi_> &= \left\{ ((\omega, n), (\omega, n)) \mid \omega \in \{a, b\}^*, n \in \mathbb{Z}, n > 0 \right\} \\
\phi_< &= \left\{ ((\omega, n), (\omega, n)) \mid \omega \in \{a, b\}^*, n \in \mathbb{Z}, n < 0 \right\} \\
\phi_+ &= \left\{ ((a\omega, n), (\omega, n+1)) \mid \omega \in \{a, b\}^*, n \in \mathbb{Z} \right\} \\
\phi_- &= \left\{ ((b\omega, n), (\omega, n-1)) \mid \omega \in \{a, b\}^*, n \in \mathbb{Z} \right\}
\end{aligned}$$

The relations can be interpreted as follows

- ϕ_0, $\phi_>$, and $\phi_<$: check whether the current integer is equal to zero, greater than zero or less than zero respectively.
- ϕ_+: if the first symbol is a, consume the first symbol and increment the current integer value.
- ϕ_-: if the first symbol is b, consume the first symbol and decrement the current integer value.

δ is defined as follows

$$\begin{aligned}
q_0\phi_- &= q_< \\
q_0\phi_+ &= q_> \\
q_<\phi_- &= q_< \\
q_<(\phi_+;\phi_<) &= q_< \\
q_<(\phi_+;\phi_0) &= q_0 \\
q_>\phi_+ &= q_> \\
q_>(\phi_-;\phi_>) &= q_> \\
q_>(\phi_-;\phi_0) &= q_0
\end{aligned}$$

A visual representation of M can be found in Figure 4.1.3. The behavior of M can be represented by a regular expression:

$$\mathcal{L}(M) = ((\phi_-(\phi_- + (\phi_+;\phi_<))^*(\phi_+;\phi_0)) + (\phi_+(\phi_+ + (\phi_-;\phi_>))^*(\phi_-;\phi_0)))^*$$

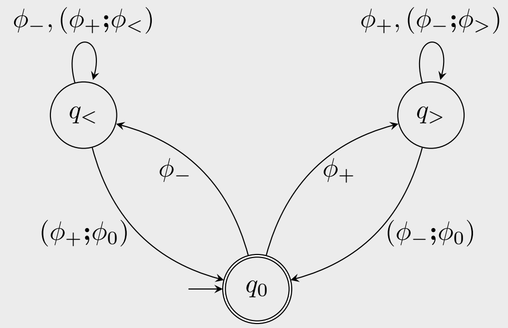

Figure 4.1.3: Visual representation of Example 4.1.3.

Consider the binary relations monoid $(\mathcal{P}(X^2), ;, id_X)$. Where $\mathcal{P}(X^2)$ is the set of all binary relations on X, $;$ is the relational composition operator (where $x\phi_1 y \wedge y\phi_2 z \implies x\phi_1;\phi_2 z$), and id_X is the identity relation on X (defined as $id_X = \{ (x, x) \mid x \in X \}$), which is the identity element of the monoid.

Suppose we have $\Phi \subseteq \mathcal{P}(X^2)$. One can observe that each element $\omega \in \Phi^*$, where Φ^* is free monoid with base Φ, can be mapped to a single binary relation by interpreting the free monoid as the binary relations monoid. This mapping is defined as follows.

Definition 4.1.2 Suppose we have $\Phi \subseteq \mathcal{P}(X^2)$, and $\omega \in \Phi^*$. $\rho_\omega \subseteq X^2$ is defined as

$$\rho_\omega = \begin{cases} \phi;\rho_{\omega'} & \text{if } \omega = \phi\omega' \\ id_X & \text{if } \omega = \varepsilon \end{cases}$$

End of Definition

The relations corresponding to the labels in the behavior of some X-machine kernel can then be defined in the following way, as discussed in [19].

Definition 4.1.3 The *characteristic relation* of a X-machine kernel M, which is binary relation on X, is defined as

$$\mathcal{C}(M) = \bigcup_{\omega \in \mathcal{L}(M)} \rho_\omega$$

End of Definition

With the full definition of the kernel, the complete X-machine is defined as follows.

Definition 4.1.4 An X-machine consists of

- the kernel M,
- an input relation $\phi_{in} : Y \to X$, where Y is some input domain,
- and an output relation $\phi_{out} : X \to Z$, where Z is some output domain.

End of Definition

The input relation *feeds* the machine an initial value from X from a value from Y. The output relation *interprets* the value from X 'computed' by the machine as a value from Z. This concept leads to the following definition.

Definition 4.1.5 A X-machine M is said to *compute* the relation $\mathcal{F}_M$ of type

$$Y \xrightarrow{\phi_{in}} X \xrightarrow{\mathcal{C}(M)} X \xrightarrow{\phi_{out}} Z$$

which is defined as

$$\mathcal{F}_M = \phi_{in};\mathcal{C}(M);\phi_{out}$$

End of Definition

Example 4.1.4 Suppose we have machine M as given in Example 4.1.1. Observe that we have

$$\mathcal{C}(M) = \phi_a;\phi_b \cup \phi_a;\phi_b;\phi_c;\phi_b \cup \phi_a;\phi_b;\phi_c;\phi_b;\phi_c;\phi_b \cup \ldots$$

We take $\{\,a, b, c\,\}^*$ as the input domain and $\mathbb{B}$ as the output domain. As our input relation we take $\phi_{in} = id_{\{\,a,b,c\,\}^*}$. As our output relation we take

$$\phi_{out}(\omega) = \begin{cases} \mathsf{true} & \text{if } \omega = \varepsilon \\ \mathsf{false} & \text{otherwise} \end{cases}$$

We now have the relation $\mathcal{F}_M : \{\,a, b, c\,\}^* \to \mathbb{B}$ computed by M. Recall automaton A as given in Example 3.1.1. Observe that we now have for each $\omega \in \{\,a, b, c\,\}^*$

$$\omega \in \mathcal{L}(A) \Leftrightarrow \mathsf{true} \in \mathcal{F}_M(\omega)$$

As an example we can take $abcb \in \mathcal{L}(A)$. Observe that $\phi_a\phi_b\phi_c\phi_b \in \mathcal{L}(M)$ and $\phi_a;\phi_b;\phi_c;\phi_b(abcb)$ $= \phi_b;\phi_c;\phi_b(bcb) = \phi_c;\phi_b(cb) = \phi_b(b) = \varepsilon$. This means that $\varepsilon \in \mathcal{C}(M)(abcb)$ which implies $\mathsf{true} \in \mathcal{F}_M(abcb)$.

Example 4.1.5 Suppose we have machine M as given in Example 4.1.2. Observe that we have

$$\mathcal{C}(M) = \phi_0 \cup \phi_*;\phi_-;\phi_0 \cup \phi_*;\phi_-;\phi_*;\phi_-;\phi_0 \cup \ldots$$

We take $\mathbb{Z}$ as both the input and output domain. As our input relation we take

$$\phi_{in} = \{\,(n, (n, 1)) \mid n \in \mathbb{Z}\,\}$$

and for our output relation

$$\phi_{out} = \{\,((n, m), m) \mid n \in \mathbb{Z}\,\}$$

We now have the relation $\mathcal{F}_M : \mathbb{Z} \to \mathbb{Z}$ computed by M. Observe that for all $n \geq 0$, we have $\mathcal{F}_M(n) = n!$ As an example we take $n = 3$. Observe that $\phi_*\phi_-\phi_*\phi_-\phi_*\phi_-\phi_0 \in \mathcal{L}(M)$, $\phi_{in}(3) = (3, 1)$ and

$$\begin{aligned}
\phi_*;\phi_-;\phi_*;\phi_-;\phi_*;\phi_-;\phi_0(3,1) &= \phi_-;\phi_*;\phi_-;\phi_*;\phi_-;\phi_0(3,3) \\
&= \phi_*;\phi_-;\phi_*;\phi_-;\phi_0(2,3) \\
&= \phi_-;\phi_*;\phi_-;\phi_0(2,6) \\
&= \phi_*;\phi_-;\phi_0(1,6) \\
&= \phi_-;\phi_0(1,6) \\
&= \phi_0(0,6) \\
&= (0,6)
\end{aligned}$$

We then have $\phi_{out}((0,6)) = 6$, which implies $\mathcal{F}_M(3) = 6$.

Example 4.1.6 Suppose we have machine M as given in Example 4.1.3. Observe that we have

$$\begin{aligned}\mathcal{C}(M) = id_{\{a,b\}^*} &\cup \phi_-;(\phi_+;\phi_0)\\ &\cup \phi_+;(\phi_-;\phi_0)\\ &\cup \phi_-;\phi_-;(\phi_+;\phi_<);(\phi_+;\phi_0)\\ &\cup \phi_+;\phi_+;(\phi_-;\phi_>);(\phi_-;\phi_0)\\ &\ldots\end{aligned}$$

We take $\{a,b\}^*$ as the input domain and $\mathbb{B}$ as the output domain. As our input relation we take

$$\phi_{in} = \{ (\omega, (\omega, 0)) \mid \omega \in \{a,b\}^* \}$$

and for our output relation

$$\phi_{out}((\omega, n)) = \begin{cases} \mathsf{true} & \text{if } \omega = \varepsilon \\ \mathsf{false} & \text{otherwise} \end{cases}$$

We now have the relation $\mathcal{F}_M : \{a,b\}^* \to \mathbb{B}$ computed by M. Observe that for all $\omega \in \{a,b\}^*$, $\mathsf{true} \in \mathcal{F}_M(\omega)$ if and only if the number a's is equal to the number of b's in ω. As an example we take $\omega = baaabb$. Observe that $\phi_-(\phi_+;\phi_0)\phi_+\phi_+(\phi_-;\phi_>)(\phi_-;\phi_0) \in \mathcal{L}(M)$ and $\phi_{in}(baaabb) = (baaabb, 0)$.

$$\begin{aligned}\phi_-;(\phi_+;\phi_0);\phi_+;\phi_+;(\phi_-;\phi_>);(\phi_-;\phi_0)((baaabb),0) &= (\phi_+;\phi_0);\phi_+;\phi_+;(\phi_-;\phi_>);(\phi_-;\phi_0)((aaabb),-1)\\ &= \phi_+;\phi_+;(\phi_-;\phi_>);(\phi_-;\phi_0)((aabb),0)\\ &= \phi_+;(\phi_-;\phi_>);(\phi_-;\phi_0)((abb),1)\\ &= (\phi_-;\phi_>);(\phi_-;\phi_0)((bb),1)\\ &= (\phi_-;\phi_0)((b),1)\\ &= (\varepsilon,0)\end{aligned}$$

This means that $(\varepsilon, 0) \in \mathcal{C}(M)((baaabb, 0))$, which implies $\mathsf{true} \in \mathcal{F}_M(baaabb)$.

4.2 Interpretation

A path in M can be interpreted as a sequence of operations on some initial value $x_0 \in X$. Suppose we have the following path p in $M = (Q, I, T, \delta)$:

$$q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \ldots \xrightarrow{\phi_n} q_n$$

where $q_0 \in I$ and $q_n \in T$. Note that $\phi_1\phi_2\ldots\phi_n \in \mathcal{L}(M)$. Suppose we have, for some $x_0 \in X$, $x_0\phi_1x_1$, $x_1\phi_2x_2$, ..., $x_{n-1}\phi_nx_n$, then we have for p

$$(q_0, x_0) \xrightarrow{\phi_1} (q_1, x_1) \xrightarrow{\phi_2} \ldots \xrightarrow{\phi_n} (q_n, x_n)$$

where for each $(q_{i-1}, x_{i-1}) \xrightarrow{\phi_i} (q_i, x_i)$ we have $q_{i-1} \xrightarrow{\phi_i} q_i$ and $x_{i-1}\phi_ix_i$. Since $x_0\phi_1x_1$, $x_1\phi_2x_2$, ..., $x_{n-1}\phi_nx_n$, we know that $x_0\phi_1;\phi_2;\ldots;\phi_nx_n$, and since $\phi_1\phi_2\ldots\phi_n \in \mathcal{L}(M)$, we now know that $x_0\mathcal{C}(M)x_n$. For this reason we can say that p forms the sequence of operations on the initial value x_0 which has as result x_n.

Chapter 5

Discrete Event Systems

In this chapter, the basic concepts of discrete event systems and supervisory control are discussed. Two existing formalisms used for modeling discrete event systems are also be briefly discussed. For these two formalisms, we also discuss supervisory control, and the algorithms regarding supervisory control synthesis for the two formalisms.

5.1 General Concepts

A *Discrete Event System* (DES) as introduced in [9, p. 31] is a *discrete-state*, *event-driven* system. This means that the state space of a DES is a discrete set. In this state set, there is a subset of *initial states* (in one of which the system will start), and a subset of *marked states*. A marked state is a state which is considered safe and stable in practical situations. The state transitions (transitions from and to states in this discrete set) are driven by *events*. An *event* occurring in a DES can correspond to an action taken by a user (e.g. a user presses a button), a condition that is met, or the activation of some actuator (e.g. a motor or a light). The set of events is partitioned into two disjoint subsets:

- *Controllable events*, which the system can prevent from happening (in practical situations these would be events corresponding with turning actuators on or off), and
- *Uncontrollable events*, which the system cannot prevent from happening (which would be events corresponding to a user interaction, a condition that is met, or a fault event).

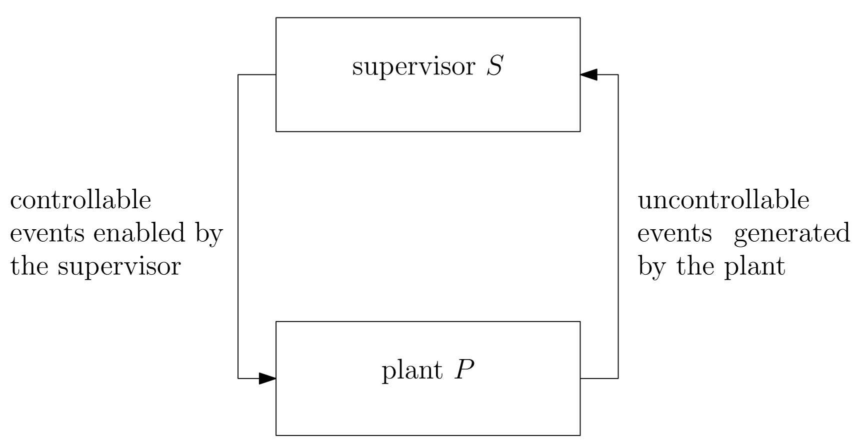

Figure 5.1.1: Supervisory control loop of the system [22, p. 96]

As discussed in [18], a system's model must consider all physical capabilities of a system, and what behavior is of the system is allowed. The following two DES must be modeled for a system to achieve this.

- A *plant*, which is a DES modeling the physically possible behavior and environment interactions of the system to be controlled.
- The *requirements* in the form of a DES, which models all allowed behavior of the system.

In the formalism for DES discussed in [18], the plant can be *refined* with respect to the requirements, which means that undesired behavior is removed from the plant. However, the resulting *refined plant* can have some undesirable properties, such as the occurrence of *blocking*. When a system is blocking, it can enter a state from which all marked states are unreachable.

A *supervisor* can prevent the system from getting to these undesired states by disabling certain controllable events (which prevents these events from happening). In this control loop, a supervisor S enables or disables controllable events based on the uncontrollable events 'generated' by the plant P (Figure 5.1.1). A *proper supervisor* assures that the following conditions are met:

- The system can always transition into a marked state, for which we say the system is *non-blocking*.
- The system does not block uncontrollable events enabled by the plant, for which we say the system is *controllable*.

A *maximally permissive supervisor* is a proper supervisor that restricts the behavior of the plant as little as possible. Computing a maximally permissive supervisor is called *supervisory control synthesis*.

In Sections 5.2 and 5.3 previously introduced formalisms for modeling discrete event systems are discussed. In Chapter 6, a formalism for modeling discrete event systems based on X-machines is introduced.

5.2 Discrete Event Systems as FSAs

The Finite State Automaton formalism is an elementary formalism (similar to the automaton formalism discussed in Chapter 3) based on automata theory which can be used for modeling discrete event systems. The simplicity of this formalism makes it a good starting point for discussing DES models and supervisory control. First, the model and its features are briefly discussed. Subsequently, a supervisory control synthesis algorithm for FSAs is discussed.

5.2.1 FSA formalism

Definition 5.2.1 A Finite State Automaton (FSA) as introduced in [9, pp. 100-120] is defined as 5-tuple $(L, \Sigma, \rightarrow, L_m, L_0)$ where

- L is the set of *locations*.
- Σ is the set of *events*.
- $\rightarrow \subseteq (L \times \Sigma \times L)$ is the *transition relation*.
- $L_m \subseteq L$ is the set of *marked locations*.

- $L_0 \subseteq L$ is the set of *initial locations*.

End of Definition

The FSA model is similar to Σ-automaton discussed in Chapter 3, where states correspond to locations and the alphabet corresponds to the event set. In the FSA model, we call the set of event sequences belonging to successful paths (starting in an initial locations and ending in a marked location) of some FSA A the *language* $\mathcal{L}(A)$, which is equivalent to the notion of *behavior* discussed in Chapter 3.

An important concept for modeling discrete event systems is *synchronization*. Synchronization allows one to break up a complex system into several simpler components, model each component separately, and combine (synchronize) these simpler components using the *synchronous product* operator. The synchronous product $||$ on two FSAs is defined in [22, pp. 57-67] as follows.

Definition 5.2.2 Suppose we have two FSA $A_1 = (L_1, \Sigma_1, \rightarrow_1, L_m^1, L_0^1)$ and $A_2 = (L_2, \Sigma_2, \rightarrow_2, L_m^2, L_0^2)$. We then have $A_1 \mid\mid A_2 = (L_1 \times L_2, \Sigma_1 \cup \Sigma_2, \rightarrow, L_m^1 \times L_m^2, L_0^1 \times L_0^2)$ where we have for $\rightarrow$

- For $\sigma \in \Sigma_1 \cap \Sigma_2$ we have $(l_1, \sigma, l_1') \in \rightarrow_1 \wedge (l_2, \sigma, l_2') \in \rightarrow_2 \iff ((l_1, l_2), \sigma, (l_1', l_2')) \in \rightarrow$.
- For $\sigma \in \Sigma_1 \setminus \Sigma_2$ we have $(l_1, \sigma, l_1') \in \rightarrow_1 \iff \forall l_2 \in L_2 : ((l_1, l_2), \sigma, (l_1', l_2)) \in \rightarrow$.
- For $\sigma \in \Sigma_2 \setminus \Sigma_1$ we have $(l_2, \sigma, l_2') \in \rightarrow_2 \iff \forall l_1 \in L_1 : ((l_1, l_2), \sigma, (l_1, l_2')) \in \rightarrow$.

End of Definition

Example 5.2.1 This example is taken from [22, p. 59]. Suppose we have

$$A_1 = (\{ l_1^1, l_2^1 \}, \{ a, b \}, \{ (l_1^1, a, l_2^1), (l_2^1, b, l_1^1) \}, \{ l_1^1 \}, \{ l_1^1 \})$$

(Figure 5.2.1) and $A_2 = (\{ l_1^2, l_2^2 \}, \{ b, c \}, \{ (l_1^2, b, l_2^2), (l_2^2, c, l_1^2) \}, \{ l_1^2 \}, \{ l_1^2 \})$ (Figure 5.2.2).

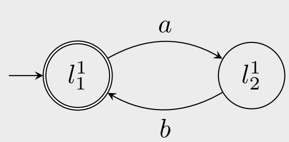

Figure 5.2.1: Visual representation of A_1

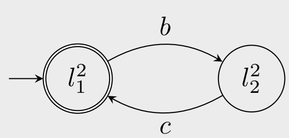

Figure 5.2.2: Visual representation of A_2

We then have the synchronous product

$$\begin{aligned}A_1 \mid\mid A_2 = (&\{(l_1^1, l_1^2), (l_2^1, l_1^2), (l_1^1, l_2^2), (l_2^2, l_2^2)\},\\ &\{a, b, c\},\\ &\{((l_1^1, l_1^2), a, (l_2^1, l_1^2)),\\ &\ ((l_2^1, l_1^2), b, (l_1^1, l_2^2)),\\ &\ ((l_1^1, l_2^2), c, (l_1^1, l_1^2)),\\ &\ ((l_1^1, l_2^2), a, (l_2^1, l_2^2)),\\ &\ ((l_2^1, l_2^2), c, (l_2^1, l_1^2))\}\\ &\{(l_1^1, l_1^2)\},\\ &\{(l_1^1, l_1^2)\},\\ &)\end{aligned}$$

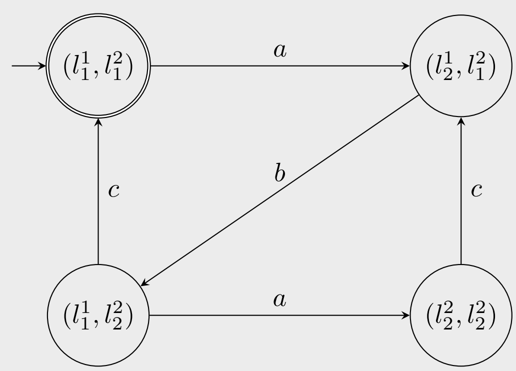

Figure 5.2.3: Visual representation of $A_1 \mid\mid A_2$

5.2.2 Supervisory Control and Synthesis Algorithm

As discussed in [22, pp. 96-118], the following properties are defined on some FSA $P = (L, \Sigma, \rightarrow, L_m, L_0)$, where Σ is partitioned into a set of controllable events $\Sigma_c \subseteq \Sigma$ and uncontrollable events $\Sigma_u \subseteq= \Sigma \setminus \Sigma_c$.

- P is *non-blocking* when for every reachable $l \in L$ (from an initial location), there exists a transition path (which may an empty path) to some $l_m \in L_m$.
- A language K is *controllable* with respect to P and uncontrollable events Σ_u if the following holds: suppose we have $\omega\omega' \in K$, $u \in \Sigma_u$, and $\omega'' \in \Sigma$ such that $\omega u \omega'' \in \mathcal{L}(P)$, then there exists some $\omega''' \in \Sigma^*$ such that $\omega u \omega''' \in K$.
- An FSA S is a *proper supervisor* for P and Σ_u when $P \mid\mid S$ is non-blocking and $\mathcal{L}(S)$ is controllable with respect to P and Σ_u.
- Proper supervisor S for P and Σ_u is *maximally permissive* when for each proper supervisor S' we have $\mathcal{L}(P \mid\mid S') \subseteq \mathcal{L}(P \mid\mid S)$.

The supervisory control problem is defined as follows: Given a plant automaton $P = (L, \Sigma, \rightarrow, L_m, L_0)$ with the sets of controllable and uncontrollable events Σ_c and Σ_u, compute a maximally permissive proper supervisor S for P and Σ_u. Algorithm 1[22, p. 118] solves the supervisory control problem for FSAs.

Algorithm 1: Supervisory Synthesis for FSA

Data: Plant $(L, \Sigma, \rightarrow, L_m, L_0)$
Result: Supervisor S
$i \leftarrow 0$;
$L^i \leftarrow L$;
do
  $N_0 \leftarrow L_m \cap L^i$;
  $N \leftarrow \texttt{FixStateSet(}L^i, N_0, \Sigma\texttt{)}$;
  $B_0 \leftarrow L^i \setminus N$;
  $B \leftarrow \texttt{FixStateSet(}L^i, B_0, \Sigma_u\texttt{)}$;
  $L^{i+1} \leftarrow L^i \setminus B$;
  $i \leftarrow i + 1$
while $L^{i-1} \neq L^i$;
$j \leftarrow 0$;
$L_s^0 \leftarrow L_0 \cap L^i$;
do
  $j \leftarrow j + 1$;
  $L_s^j \leftarrow L_s^{j-1} \cup \left\{ l \in L^i \mid l_s \xrightarrow{\sigma} l, l_s \in L_s^{j-1} \right\}$
while $L_s^{j-1} \neq L_s^j$;
return $(L_s^j, \Sigma, \rightarrow \cap (L_s^j \times \Sigma \times L_s^j), L_m \cap L_s^j, L_0 \cap L_s^j)$

Function $\texttt{FixStateSet(}L', X, \Gamma\texttt{)}$
  $i \leftarrow 0$;
  $X_0 \leftarrow X$;
  do
    $X_{i+1} \leftarrow X_i \cup \left\{ l \in L' \mid l \xrightarrow{\sigma} x, x \in X_i, \sigma \in \Gamma \right\}$;
    $i \leftarrow i + 1$;
  while $X_{i-1} \neq X_i$;
  return X_{i-1}

Example 5.2.2 This example is taken from [22, pp. 114-117]. Suppose we have plant

$$P = (\{ l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9 \}, \Sigma, \rightarrow, L_m, L_0)$$

as shown Figure 5.2.4, with $\Sigma_c = \{ c_1, c_2, c_3 \}$ and $\Sigma_u = \{ u_1, u_2 \}$. It models a workcell consisting of two machines and an automated guided vehicle. The vehicle can load and unload a workpiece at machine 1 or 2, represented by u_1, c_1, u_2, and c_2 respectively, and unload it to a buffer, represented by c_3.

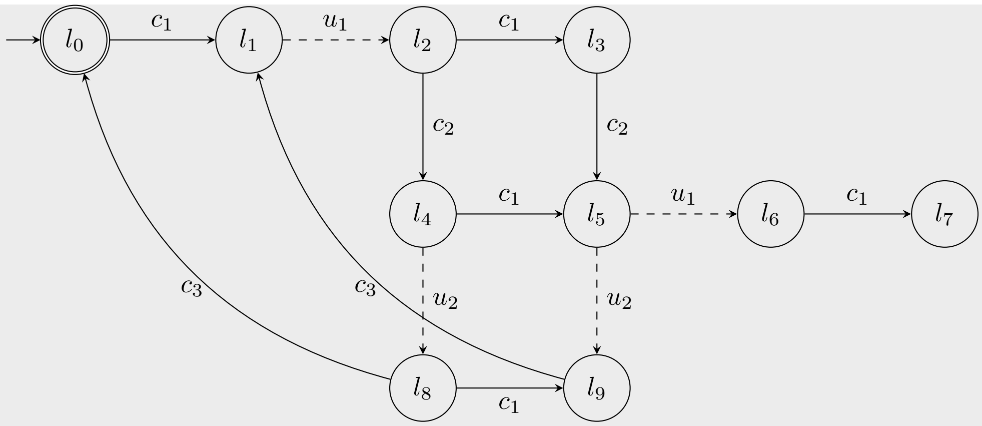

Figure 5.2.4: Visual representation of P, edges with uncontrollable events are drawn with dashed lines.

We will now compute the maximally permissive supervisor of P using Algorithm 1:
We start with $L^0 = \{ l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9 \}$. For the first (outer) do-while loop we have the following iterations.

Iteration 1 $N_0 = \{ l_0 \}$
$N = \{ l_0, l_8, l_4.l_2, l_1, l_9, l_5, l_3 \}$
$B_0 = \{ l_6, l_7 \}$
$B = \{ l_6, l_7, l_5 \}$

$L^1 = \{ l_0, l_1, l_2, l_3, l_4, l_8, l_9 \}$.

Iteration 2 $N_0 = \{ l_0 \}$
$N = \{ l_0, l_8, l_4, l_2, l_1, l_9 \}$
$B_0 = \{ l_3 \}$
$B = \{ l_3 \}$

$L^2 = \{ l_0, l_1, l_2, l_4, l_8, l_9 \}$

Iteration 3 $N_0 = \{ l_0 \}$
$N = \{ l_0, l_8, l_4, l_2, l_1, l_9 \}$
$B_0 = \emptyset$
$B = \emptyset$

$L^3 = L^2$ The first for-loop will terminate since $L^3 = L^2$.

For the second do-while loop we end up with $L_s^j = L^2$ (since every location in L^2 is reachable).
We end up with supervisor $(L_s^j, \Sigma, \rightarrow \cap (L_s^j \times \Sigma \times L_s^j), L_m \cap L_s^j, L_m \cap L_s^j)$ (Figure 5.2.5).

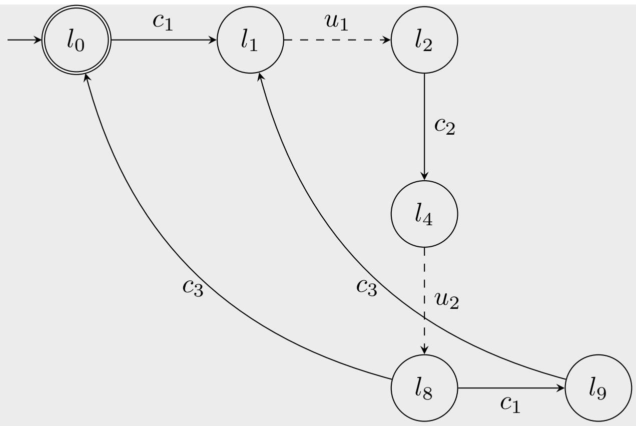

Figure 5.2.5: Visual representation of P.

5.3 Discrete Event Systems as EFAs

The Extended Finite Automaton model is an extension of the FSA model. In EFAs part of the system's state space is modeled as some finite domain D. This means that, for example, some system's state space is not only described using a locations set L, but a location set along with a set of integer values (for example $D = \{0, \ldots, n\}$). Guards and update functions are added to the transitions edges, to allow transitions between instances of the domain. This extension allows for a more efficient way of modeling discrete event systems, which is discussed in [18].

5.3.1 EFA Formalism

We discuss the EFA model as discussed in [18, p.2].

Definition 5.3.1 An Extended Finite Automaton (EFA) is defined as 7-tuple $(L, D, \Sigma, E, L_0, D_0, L_m)$ where the elements additional to FSAs are defined as follows

1. $D = D_1 \times \cdots \times D_p$ is a domain of data values consisting of p 'variables',
2. E is the set of *edges*,
3. $D_0 = D_0^0 \times \cdots \times D_0^p$ is the set of initial data values.

For every edge $e \in E$ we have

- $o_e \in L$ and $t_e \in L$ are the origin and target locations of the edge,
- $\sigma_e \in \Sigma$ is the event of the edge,
- $g_e \subseteq D$ is the enabling guard of the edge,
- $f_e : D \to D$ is the update function of the edge.

End of Definition

As discussed in [24], it must be noted that any EFA A can be reduced to an FSA A' by eliminating the domain D, where D is essentially reduced to extra state space. This means that A' has location set $L' = L \times D$ (L is the location set of A). For each transition edge e in A, there is transition $(o_e, d) \xrightarrow{\sigma_e} (t_e, f_e(d'))$ for each $d \in g_e$ in A'.

As for FSA, a synchronous product operator is defined for EFAs in [18, pp. 2-3]

Definition 5.3.2 Suppose we have EFA $A_1 = (L_1, D_1, \Sigma_1, E_1, L_0^1, D_0^1, L_m^1)$ and $A_2 = (L_2, D_2, \Sigma_2, E_2, L_0^2, D_0^2, L_m^2)$ the synchronous product $A_1 \mid\mid A_2$ is defined as follows

$$A_1||A_2 = (L_1 \times L_2, D_1 \otimes D_2, \Sigma_1 \cup \Sigma_2, E, L_0^1 \times L_0^2, D_0^1 \otimes D_0^2, L_m^1 \times L_m^2)$$

Suppose there is a given domain composition $D_1 \otimes D_2 = D_1' \times D_s \times D_2'$ where $D_1 = D_1' \times D_s$ and $D_2 = D_2' \times D_s$ (D_s is shared between the two domains in this given domain composition). The set of edges E is defined as follows:

- $\forall \sigma \in E_1 \cap E_2, \forall (l_1, l_2, \sigma, g_1, f_1) \in E_1, \forall (l_2, l_2', \sigma, g_2, f_2) \in E_2$, we have $((l_1, l_2), (l_1', l_2'), \sigma, g_1 \wedge g_2 \wedge [f_1|_{D_s} = f_2|_{D_s}], f_1 \oplus f_2) \in E$.
- $\forall \sigma \in \Sigma_1 \setminus \Sigma_2, \forall (l_1, l_1', \sigma, g_1, f_1) \in E_1$ we have $\forall l_2 \in L_2$, $((l_1, l_2), (l_1', l_2), \sigma, g_1, f_1) \in E$.
- $\forall \sigma \in \Sigma_2 \setminus \Sigma_1, \forall (l_2, l_2', \sigma, g_2, f_2) \in E_2$ we have $\forall l_1 \in L_1$, $((l_1, l_2), (l_1, l_2'), \sigma, g_2, f_2) \in E$.

where '$f_1 \oplus f_2 : D_1 \otimes D_2 \to D_1 \otimes D_2$ maps the shared data variables D_s identically as either of the functions map, whereas it maps the nonshared data variables according to the functions whose domain they belong'[18].
End of Definition

Example 5.3.1 Suppose we have EFA

$$A_1 = (\{ l_0^1, l_1^1 \}, \{ \square \}, \{ a, b \}, \{ e_{0,1}, e_{1,0} \}, \{ l_0^1 \}, \{ \square \}, \{ l_0^1 \})$$

shown in Figure 5.3.1 (since we do not have a domain for A_1 we use placeholder $\square$) and

$$A_2 = (\{ l_0^2, l_1^2 \}, \{ 0, \ldots, 4 \}, \{ a, b \}, \{ e_{0,0}^1, e_{0,0}^2, e_{1,0}, e_{1,1}^1, e_{1,1}^2 \}, \{ l_1^2 \}, \{ \square \}, \{ l_1^2 \})$$

shown in Figure 5.3.2. Suppose we have $\{ 0 \} \otimes \{ 0, \ldots, 4 \} = \{ 0 \} \times \{ 0, \ldots, 4 \}$ (this means that there is no shared domain). We then have $A_1||A_2$ as shown in Figure 5.3.3.

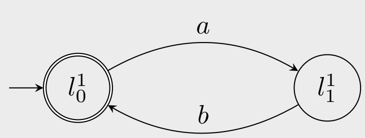

Figure 5.3.1: Visual representation of A_1.

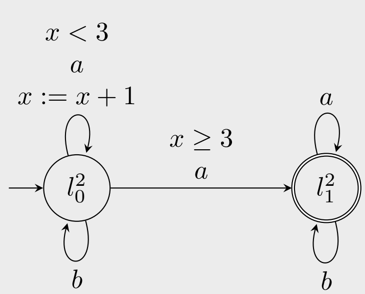

Figure 5.3.2: Visual representation of A_2, where x represents the domain value

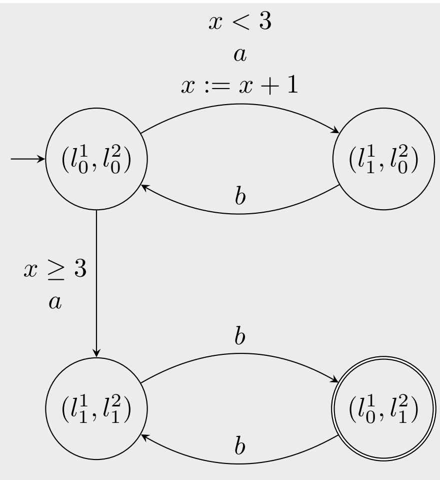

Figure 5.3.3: Visual representation of $A_1 \,||\, A_2$

5.3.2 Plants and Requirements

A model for a discrete event system consists of

- The plant $P = P_1 \,||\, P_2 \,||\, \dots$, which is an EFA modeling the physical behavior of the system. Each P_i models a physical component of the system (e.g. an actuator or a sensor).
- The requirements $R = R_1 \,||\, R_2 \,||\, \dots$, which is an EFA modeling the allowed behavior of the system (sequences of events). Each R_i models a single atomic requirement (e.g. some actuator may only activate after some button is pressed).

Given the plant P and the requirements R we can compute the *refined plant* [18, p. 3]. By refining P with respect to R we remove the unwanted behavior from the plant.

Definition 5.3.3 Suppose we have plant EFA $P = (L^P, D, \Sigma, E^P, L_0^P, D_0, L_m^P)$ and requirements EFA $R = (L^R, D, \Sigma, E^R, L_0^R, D_0, L_m^R)$. The refined plant is defined as EFA $G = (L^P \times (L^R \cup \{\phi\}), D, \Sigma, E, L_0^P \times L_0^R, D_0, L_m^P \times L_m^R)$ where E is constructed as follows:

- $\forall e \in E^P, \forall l \in L^R \cup \{\phi\}, \forall e' \in E^R$ with $(o_{e'} = l) \wedge (\sigma_{e'} = \sigma_e) : ((o_e, l), (t_e, t_{e'}), \sigma_e, g_e \wedge g_{e'} \wedge [f_e = f_{e'}], f_e) \in E, ((o_e, l), (t_e, \phi), \sigma_e, g_e \wedge \neg \left[\exists_{e'' \in E^R : o_{e''} = o_{e'}, \sigma_{e''} = \sigma_{e'}}, g_{e''} \wedge [f_{e''} = f_{e'}]\right], f_e) \in E$,
- $\forall e \in E^P, \forall l \in L^R \cup \{\phi\}, \neg \exists e' \in E^R$ with $(o_{e'} = l) \wedge (\sigma_{e'} = \sigma_e) : ((o_e, l), (t_e, \phi), \sigma_e, g_e, f_e) \in E$.

An extra location identifier ϕ is added to L^R. This is a location outside of the 'allowed' state space L^R, which means ϕ can be interpreted as a 'forbidden location'. The set of *forbidden locations* of G is then defined as $L_f = \left\{ (l, \phi) \mid l \in L^P \right\}$.

End of Definition

Example 5.3.2 This example is taken from [18, p. 4]. Suppose we have plant P as shown in Figure 5.3.4 and requirement R as shown in Figure 5.3.5. P refined with respect to R (P_R) is shown in Figure 5.3.6. For the forbidden locations we have $L_f = \{ (l_0, \phi), (l_1, \phi) \}$.

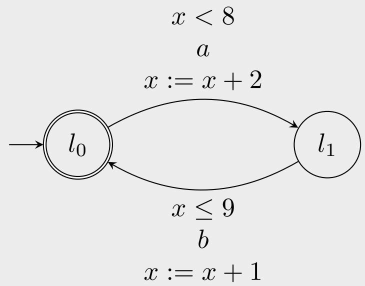

Figure 5.3.4: Visual representation of P, where x represents the domain value

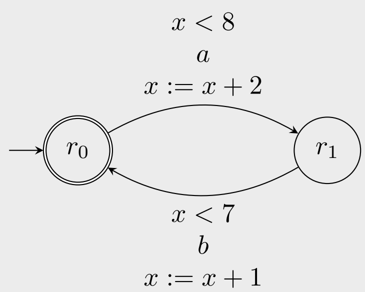

Figure 5.3.5: Visual representation of R, where x represents the domain value

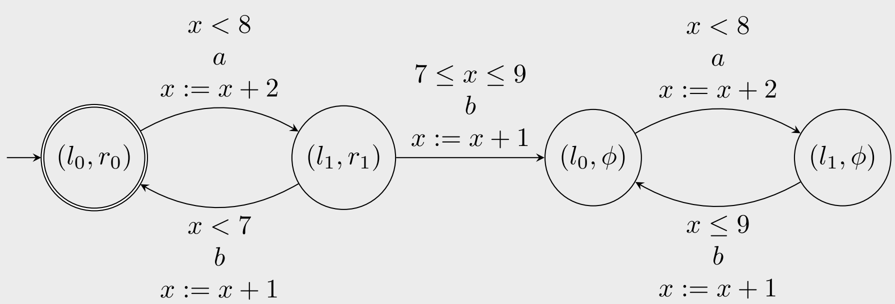

Figure 5.3.6: Visual representation of P_R, where x represents the domain value

5.3.3 Supervisory Control and Synthesis Algorithm

In this section we discuss supervisory control synthesis for a given refined plant

$$G = (L, D, \Sigma, E, L_0, D_0, L_m)$$

with a set of forbidden locations $L_f \subseteq L$. We again have the event set partitioned into the set of controllable events Σ_c and uncontrollable events Σ_u. For supervisory control synthesis the guards of G will be strengthened, for which the following definition is introduced in [18, p. 3].

Definition 5.3.4 Suppose we have refined plant $G = (L, D, \Sigma, E, L_0, D_0, L_m)$ and function $\mathcal{S} : E \to D$. We define $G^{\mathcal{S}} = (L, D, \Sigma, E', L_0, D_0, L_m)$ with

$$E' = \{\, e' \mid e \in E, e' \text{ is } e \text{ with guard replaced with } g_e \wedge \mathcal{S}(e) \,\}$$

End of Definition

In order to better reason about how much a supervisor restricts G, the following ordering on EFAs is introduced in [18, p. 3].

Definition 5.3.5 Suppose we have EFAs G and G' the we define ordering ($\preccurlyeq$) as

$$G' \preccurlyeq G$$

if and only if , G' is obtained from G by strengthening (a) guard(s), removing (an) edge(s), and/or removing (a) location(s).

End of Definition

The refined automaton G and the forbidden locations L_f are given as input for the Supervisory Synthesis algorithm, which computes a function for strengthening guards of the edges of G (a supervisor for G), such that the following conditions as introduced in [18, p. 3] hold:

1. $G^{\mathcal{S}}$ is *nonblocking*: from every state in $G^{\mathcal{S}}$, there exists some path to a marked state.
2. $G^{\mathcal{S}}$ is *safe*: no state in L_f is reachable from an initial state in $G^{\mathcal{S}}$.
3. $\mathcal{S}$ is *controllable* with respect to G if and only if there is some l and d for which there is $l \xrightarrow{e} l'$ with $d \in g_e$ (meaning that the guard g_e 'allows' the value d) in G then there is e' for which $l \xrightarrow{e'} l'$ with $d \in g_e'$ in $G^{\mathcal{S}}$.
4. $\mathcal{S}$ is a *proper supervisor* for G if and only if $G^{\mathcal{S}}$ is nonblocking and safe, and $\mathcal{S}$ is controllable with respect to G.
5. Proper supervisor $\mathcal{S}$ for G is a maximally permissive supervisor if and only if for every proper supervisor $\mathcal{S}'$ for G we have $G^{\mathcal{S}'} \preccurlyeq G^{\mathcal{S}}$.

The supervisory control on EFAs is defined as follows: Given refined plant EFA

$$G = (L, D, \Sigma, E, L_0, D_0, L_m)$$

with the event set Σ partitioned into the sets of controllable events Σ_c and uncontrollable events Σ_u, and a set of forbidden locations $L_f \subseteq L$. Compute the maximally permissive supervisor $\mathcal{S}$ for G.

Algorithm 2 from [18, p. 4] solves the supervisory control for EFA.

Algorithm 2: Supervisory Synthesis for EFAs

Data: EFA $G = (L, D, \Sigma, E, L_0, D_0, L_m)$ with set of forbidden locations $L_f \subset L$ and sets Σ_c and Σ_d (with $\Sigma_c \subseteq \Sigma$ and $\Sigma_u = \Sigma \setminus \Sigma_c$)

Result: Updated guard for every $e \in E$

$\forall e \in E : g_e^0(d) \leftarrow g_e(d)$;

$j \leftarrow 0$;

do

$\quad \forall l \in L, d \in D : N_j(l,d) \leftarrow \begin{cases} T, & \text{if } l \in L_m \\ F, & \text{if } l \notin L_m \end{cases}$;

$\quad N_j \leftarrow$ `FixPredicate(`N_j`,` Σ`)` ;

$\quad \forall l \in L, d \in D : B_j(l,d) \leftarrow \begin{cases} T, & \text{if } l \in L_f \\ \neg N_j(l,d), & \text{if } l \notin L_f \wedge j = 0 \\ \neg N_j(l,d) \vee B_{j-1}(l,d), & \text{if } l \notin L_f \wedge j > 0 \end{cases}$;

$\quad B_j \leftarrow$ `FixPredicate(`B_j`,` Σ_u`)` ;

$\quad \forall e \in E, d \in D : g_e^{j+1}(d) \leftarrow \begin{cases} g_e^j(d) \wedge \neg B_{t_e}^j(f_e(d)), & \text{if } \sigma \in \Sigma_c \\ g_e^j(d), & \text{if } \sigma \in \Sigma_u \end{cases}$;

$\quad j \leftarrow j + 1$

while $\neg \forall e \in E, d \in D : g_e^j(d) = g_e^{j-1}(d)$;

return *for all* $e \in E$: g_e^{j-1}

Function `FixPredicate(`$P : L \times D \to \mathbb{B}$`,` $\Sigma_s \subseteq \Sigma$`)`

$\quad i \leftarrow 0$;

$\quad P_0 \leftarrow P$;

$\quad$ **do**

$\qquad$ `/* Update the predicate. */`

$\qquad \forall l \in L, d \in D : P_{i+1}(l,d) \leftarrow P_i(l,d) \vee \bigvee_{e | o_e = l, \sigma_e \in \Sigma_s} \left[g_e^j(d) \wedge P_i(t_e, f_e(d)) \right]$;

$\qquad i \leftarrow i + 1$

$\quad$ **while** $\neg \forall l \in L, d \in D : P_{i-1}(l,d) = P_i(l,d)$;

$\quad$ **return** P_{i-1}

Example 5.3.3 This example is taken from [22, pp. 135-136]. Suppose we have EFA

$$P = (\{ l_0, l_1, l_2 \}, \{ 0, \ldots, 10 \}, \{ c, u \}, \{ e_{0,1}, e_{1,2}, e_{2,0} \}, \{ l_0 \}, 0, \{ l_0 \})$$

shown in Figure 5.3.7, where $\Sigma_c = \{ c \}$ and $\Sigma_u = \{ u \}$. In P we essentially have a counter which is incremented by 1 during every transition, where event u may only occur when $x < 7$. Observe that P can enter l_2 with $x \geq 7$ at which P will block.

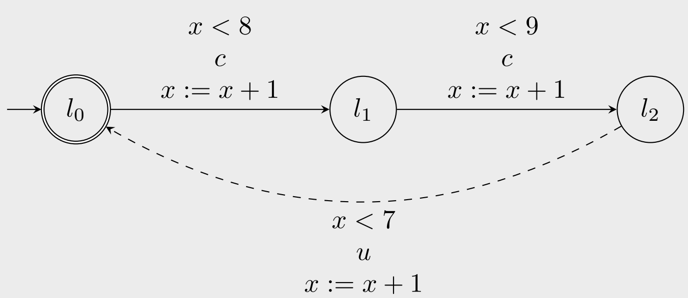

Figure 5.3.7: Visual representation of P.

We will now perform Algorithm 2 on P:
We start with $j = 0$

Iteration 1 $N_0(l, x) = \mathsf{true}$ for $l = l_0 \wedge x \in \{0, \dots, 10\}$.
$N(l, x) = \mathsf{true}$ for $l = l_0 \wedge x \in \{0, \dots, 10\}$, $l = l_2 \wedge x \in \{0, 6\}$, $l = l_1 \wedge x \in \{0, 5\}$.
$B_0(l, x) = \mathsf{true}$ for $l = l_2 \wedge x \in \{7, 10\}$, $l = l_1 \wedge x \in \{6, 10\}$.
$B = B_0$.

$g^1_{e_{0,1}} : g^0_{e_{0,1}}(x) \wedge x < 5$
$g^1_{e_{1,2}} : g^0_{e_{1,2}}(x) \wedge x < 6$

Iteration 2 $N(l, x) = \mathsf{true}$ for $l = l_0 \wedge x \in \{0, \dots, 10\}$, $l = l_2 \wedge x \in \{0, 6\}$, $l = l_1 \wedge x \in \{0, 5\}$.
$B_0(l, x) = \mathsf{true}$ for $l = l_2 \wedge x \in \{7, 10\}$, $l = l_1 \wedge x \in \{6, 10\}$.
$B = B_0$.

The bad predicate is equivalent to the bad predicate found in the previous iteration, which means the guards do not change. This implies that the algorithm terminates

P with the obtained supervisor $\mathcal{S}$ applied is shown in Figure 5.3.8. Observe that $P^{\mathcal{S}}$ cannot reach l_2 with the domain value of $x \geq 7$, which means that $P^{\mathcal{S}}$ will not block.

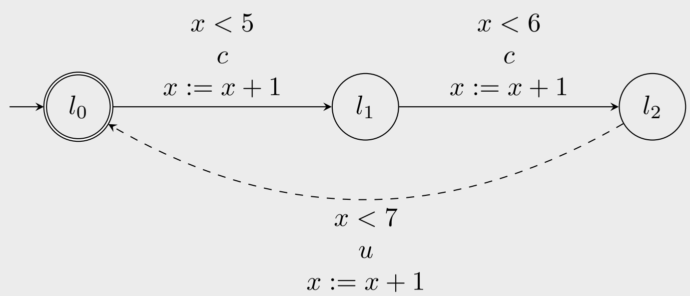

Figure 5.3.8: Visual representation of P with supervisor applied.

5.3.4 Limitations

In this section two limitations of the EFA formalism are discussed. These limitations will be our points of attention when we will define our own formalism for modeling discrete event systems in Chapter 6.

Scalability Issue

We first discuss a limitation regarding scalability. We run into this problem when we try to model a more complex 'input device'. Suppose we have a set of 'user commands' $C = \{c_1, \dots, c_n\}$. When the system is idle, a user can give a command to the device. After a command is given, the user waits for a *response* from the system. The system responds to the user after the command has been handled by the system. Intuitively our device has two events: the uncontrollable event 'inputCommand' and the controllable event 'respond'. For our domain, we take $D = \{\Box\} \cup C$, which is the set of commands together with a 'no-command' token ($\Box$). We have $D_0 = \{\Box\}$ since no command is given in the starting state. We can model the input device P_i (as part of the plant P) as follows.

$$P_i = (\{p_0, p_1\}, D, \{\text{inputCommand}, \text{respond}\}, E_p, \{p_0\}, D_0, \{p_0\})$$

For every $c \in C$ there exists edge $e \in E_p$ with

- $o_e = p_0$,
- $t_e = p_1$,
- $\sigma_e = \text{inputCommand}$,
- $g_e = D$ (no guard needed),
- $f_e(d) = c$.

An there exists $e' \in E_p$ with

- $o_{e'} = p_1$,
- $t_{e'} = p_0$,
- $\sigma_{e'} = \text{respond}$,
- $g_{e'} = D$ (no guard needed),
- $f_{e'}(c) = \Box$.

A visual representation of P_i is shown in Figure 5.3.9.

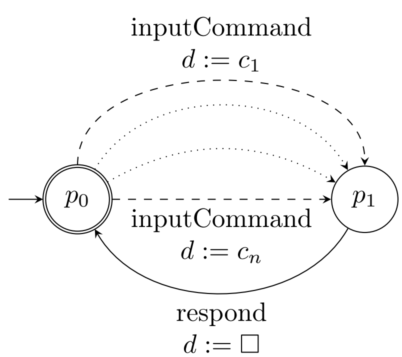

Figure 5.3.9: The input device P_i

We can now model a requirement R_j (as part of the complete requirement R) as follows.

$R_j = (\{\, r_0, r_1, \ldots, r_m \,\}, (D \times D'), \{\, \text{inputCommand}, \text{respond}, \ldots \,\}, E_r, \{\, r_0 \,\}, \{\, (\Box, d'_1), \ldots \,\}, \{\, r_0 \,\})$

This requirement has the domain D combined with other variables (if needed), and the event set contains (at least) 'inputCommand' and 'respond'. Suppose this requirement models the desired behavior after the command c_k is given, then we have for every $c \in (C \setminus \{\, c_k \,\})$ an edge $e \in E_r$ with

- $o_e = r_0$,
- $t_e = r_0$,
- $\sigma_e = \text{inputCommand}$,
- $g_e = D \times D'$ (no guard needed),

- $f_e((\square, d')) = (c, d')$.

These edges are introduced to prevent the system from blocking all other user commands. We also introduce the following self-loop edge e' in r_0 which prevents the system from blocking the event 'respond'. e' has

- $o_{e'} = r_0$,
- $t_{e'} = r_0$,
- $\sigma_{e'} =$ respond,
- $g_{e'} = D \times D'$ (no guard needed),
- $f_{e'}((c, d')) = (\square, d')$.

For the command c_k there exists the edge $e'' \in E_r$ with

- $o_{e''} = r_0$,
- $t_{e''} = r_1$,
- $\sigma_{e''} =$ inputCommand,
- $g_{e''} = D \times D'$ (no guard needed),
- $f_{e''}((\square, d')) = (c_k, d')$.

Finally we say that E_r also has edge e''' with

- $o_{e'''} = r_m$,
- $t_{e'''} = r_0$,
- $\sigma_{e'''} =$ respond,
- $g_{e'''} = D \times D'$ (no guard needed),
- $f_{e'''}((c_k, d')) = (\square, d')$.

A visual representation of R_j is shown in Figure 5.3.10.

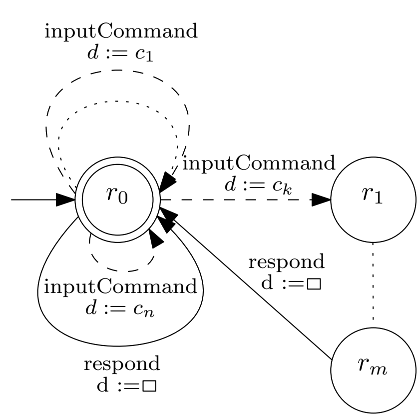

Figure 5.3.10: The requirement R_j

One can observe modeling the input devices using the method we just discussed is rather inconvenient. We have to create a transition edge for each command both in P_i and all requirements considering the input device. Also, for each edge a function has to be defined.

An alternative way to model this input is to break up the event 'inputCommand' in n events $c_1, \ldots, c_n$ (meaning that we introduce a separate event for each command). This will result in the following model.

$$P'_i = (\{ p_0, p_1 \}, \{ \square \}, \{ c_1, \ldots, c_n, \text{respond} \}, E_p, \{ p_0 \}, \{ \square \}, \{ p_0 \})$$

The domain $\{ \square \}$ is just a place holder (we do not need a domain in this case). For every $c_k \in C$ we have $e \in E_p$ with

- $o_e = p_0$,
- $t_e = p_1$,
- $\sigma_e = c_k$,
- $g_e = \{ \square \}$,
- $f_e(\square) = \square$.

And there exists edge $e' \in E_p$ with

- $o_{e'} = p_1$,
- $t_{e'} = p_0$,
- $\sigma_{e'} = \text{respond}$,
- $g_{e'} = \{ \square \}$,
- $f_{e'}(\square) = \square$.

We can then model a requirement R'_j as follows. Again, R'_j only considers the command c_k.

$$R'_j = (\{ r_0, r_1, \ldots, r_m \}, D', \{ c_k, \text{respond}, \ldots \}, E_r, \{ r_0 \}, D'_0, \{ r_0 \})$$

Where we again have a self loop transition edge $e \in E_r$ to prevent the blocking of 'respond'.

- $o_{e'} = r_0$,
- $t_{e'} = r_0$,
- $\sigma_{e'} = \text{respond}$,
- $g_{e'} = D'$,
- $f_{e'}(d') = d'$.

For the command c_k there exists an edge $e'' \in E_r$ with

- $o_{e''} = r_0$,
- $t_{e''} = r_1$,
- $\sigma_{e''} = c_k$,
- $g_{e''} = D'$,

- $f_{e''}(d') = d'$.

And we say that E_r also has edge e''' with

- $o_{e'''} = r_m$,
- $t_{e'''} = r_0$,
- $\sigma_{e'''} = \text{respond}$,
- $g_{e'''} = D'$,
- $f_{e'''}(d') = d'$.

A visual representation of R'_j is shown in Figure 5.3.11.

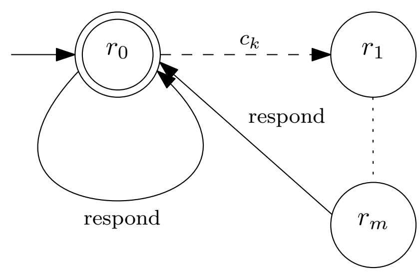

Figure 5.3.11: The requirement R'_j

Essentially, we have modeled our input device as an 'FSA' (since P'_i does not have a domain). One can observe that modeling requirements is more convenient using this method. However, we now have to create an event for every command, and we still have to create a transition edge for every command in P_i.

We can observe that, using the EFA formalism, we can run into these kinds of problems when dealing with more complex input devices. In order to model the user giving some input as a single event, we need a formalism in which the following is possible.

- The method of 'the user giving input' can be incorporated in the event itself.
- Transitions can be enabled/disabled (in requirement models) depending on the 'result' of the event. In this case, that result would be the specific command given by the user.

Physical Restrictions

Another limitation of this formalism is that it is difficult to deal with restrictions with a plant P caused by physical relations between separate subcomponents of P. For example, suppose P has subcomponent $P_i = (L_i, D_i, \Sigma_i, E_i, L_0^i, D_0^i, L_m^i)$ and $P_j = (L_j, D_j, \Sigma_j, E_j, L_0^j, D_0^j, L_m^j)$, where there is a location $l \in L_i$ which *restricts* event $\sigma \in \Sigma_j$. This could be the case when some light source is switched on in l, and σ is an event corresponding to a light sensor, located near the aforementioned light source, giving a 'low' signal. Since this is not possible in a physical situation, we do not want this behavior (the sensor given a 'low' signal when the light source is on) in our plant P. However, since P_i and P_j are independent components outside the context of the system, we cannot implement this restriction in P_i or P_j. This means we need an extra operation on P to implement these kinds of restrictions in our plant.

Chapter 6

Discrete Event Systems as D-Systems

In this chapter, a formalism is introduced for modeling discrete event system based on X-Machines. The following items are discussed:

- a new concept of 'events' which will be used in our formalism,
- the introduction of the D-system formalism and its components, which can be used to model discrete event systems,
- synchronization of D-systems by introduction of a synchronous product operator,
- the algebraic properties of the synchronous product operator,
- a method for modeling a plant and requirements using D-systems,
- a restriction operator for D-systems, which can be used to restrict the behavior of an existing plant,
- the definitions regarding supervisory control for D-systems, and an algorithm to compute a most permissive supervisor.

By introducing this formalism for modeling discrete event systems, we can address the limitations of the EFA formalism discussed in Subsection 5.3.4. By tackling these issues when defining the formalism, we evade them during the language design in part III.

6.1 Conceptual Background

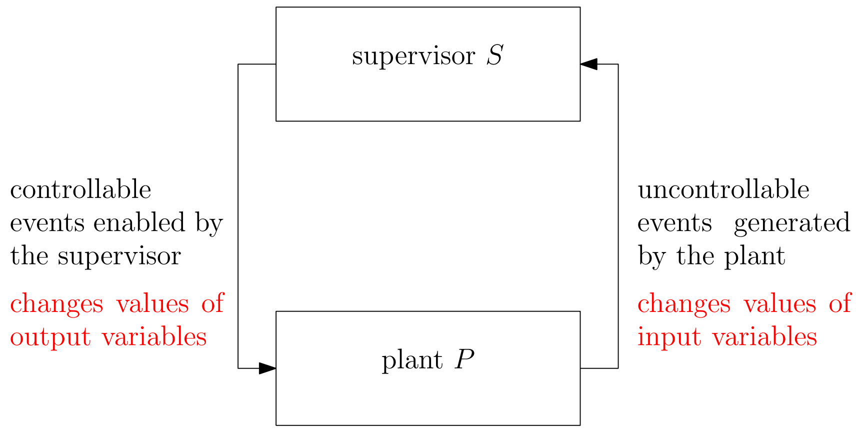

Figure 6.1.1: Supervisory control loop with input/output.

In order to model discrete event systems in some formalism, we need to define a concrete concept of an *'event'*. In the EFA model, events are modeled by the elements of the alphabet Σ. From supervisory control theory, we have that events are either controllable (an event which is 'initiated' by the system) or uncontrollable (event which is received by the system). For EFAs, the Σ is partitioned between uncontrollable event set Σ_u and Σ_c.

In practice, an event (in some system) corresponds to an input/output change within the system. A controllable event corresponds to a change of 'output values' (actuators) of the system, and an uncontrollable event corresponds to a change of 'input values' (sensors) of the system (as shown in Figure 6.1.1). We base our definition of event for X-machines on this concept.

6.2 Definitions

In this section we give a definition for *D-systems*, which is our new formalism for defining discrete event system based on X-machines. We first discuss the different components which our formalism should have. Then, we define and discuss these different components. Lastly, we give our formal definition of *D-system*.

6.2.1 System Components

Our formalism for discrete event systems should have the following components.

- An underlying domain D.
- Two disjoint sets of controllable and uncontrollable events which are sets of relations on domain D. We discuss the concrete concept of an *event* in Section 6.2.2.
- The underlying D-machine, which transitions should contain the aforementioned events. We discuss our implementation of D-machines within D-systems in Section 6.2.3.
- The set of possible initial values of the system, which is a subset of D. This we also discuss in Section 6.2.3.

6.2.2 Events

We now introduce a definition based on the concept of events discussed in Section 6.1.

Definition 6.2.1 We define an *event* in some system with domain D as a binary relation on D. The set of events is partitioned into the two disjoint sets of controllable events E_c and uncontrollable events E_u.

End of Definition

A relation corresponding to a controllable event models (a) value change(s) of some output variable(s) of the system, and a relation corresponding to an uncontrollable event models (a) change(s) of some input variable(s) of the system.

Example 6.2.1 Suppose we have some system with a domain $D = \mathbb{B}_a \times \mathbb{B}_s$ where the first Boolean value models the on/off status of an actuator and the second Boolean value models the on/off status of a sensor. Suppose the system has controllable events $E_c = \{\, \text{actuator\_switch} \,\}$ and uncontrollable events $E_u = \{\, \text{sensor\_switch} \,\}$. The definition of the events are as follows:

$$\begin{aligned} \text{actuator\_switch} &= \{\, ((b, b), (\neg b, b)) \mid (b, b) \in D \,\} \\ \text{sensor\_switch} &= \{\, ((b, b), (b, \neg b)) \mid (b, b) \in D \,\} \end{aligned}$$

The event actuator_switch models switching the actuator on/off, and the event sensor_switch models the change of detection of the sensor.

6.2.3 EventMachines

In this subsection, we will incorporate X-machines (as discussed in Chapter 4) in our D-system formalism. In our formalism we should be able to do the following.

- Guard transitions: This means that one can specify for which domain values a transition is enabled. Being able to guard transitions is needed to disable controllable events, which is important for supervisory control synthesis.
- Interpret events: Since the events are modeled as binary relations, there could be multiple possible outcomes for an event (for one instance of the domain value). One should be able to enable or disable a transition based on the possible outcomes of its event.

To achieve this, we introduce *D-EventMachines*, which is a slight modification to the vanilla X-machine discussed in Chapter 4.

Definition 6.2.2 Suppose we have some system with domain D with controllable events E_c and uncontrollable events E_u. Then the behavior of this system is modeled by a *D-EventMachine* which is defined as a 5-tuple (Q, I, T, Φ, δ). This definition is equivalent to the definition of the X-machine kernel from Definition 4.1.1, with the exception that instead of $\Phi \subseteq \mathcal{P}(D^2)$ we now have

$$\Phi \subseteq \mathcal{P}(D^2) \times (E_c \cup E_u) \times \mathcal{P}(D^2)$$

This means that every element of Φ is a 3-tuple (ϕ, e, ϕ'), where ϕ and ϕ' can be any arbitrary binary relations on D (meaning $\phi, \phi' \in \mathcal{P}(D)$) and e is an event (meaning $e \in E_c \cup E_u$).

Suppose we have some event $e \in (E_c \cup E_u)$. We then say event e occurs if and only if some transition $q \xrightarrow{(\phi, e, \phi')} q'$ occurs for some $q, q' \in Q$ and $(\phi, e, \phi') \in \Phi$.

Suppose we have the following path p

$$(q_0, d_0) \xrightarrow{(\phi_1, e_1, \phi_1')} (q_1, d_1) \xrightarrow{(\phi_2, e_2, \phi_2')} \cdots \xrightarrow{(\phi_n, e_n, \phi_n')} (q_n, d_n)$$

We then say that p can occur in the EventMachine if and only if $q_0 \in I$, $q_n \in T$, $\forall_{0<i\leq n} q_{i-1}(\phi_i, e_i, \phi_i') \ni q_i$ (as in the transition $q_{i-1} \xrightarrow{(\phi_i, e_i, \phi_i')} q_i$ can occur), and $d_{i-1}(\phi;e;\phi')d_i$.

End of Definition

We introduce a definition which helps us reason about the trace sets produced by D-EventMachines.

Definition 6.2.3 For each $t \in \Phi$ with $t = (\phi, e, \phi')$ we define the *label relation* $\rho_t \subseteq D^2$ as

$$\rho_t = \phi;e;\phi'$$

By extension, for each $\omega \in \Phi^*$ with $\omega = t_1 \dots t_n$ we define $\rho_\omega \subseteq D^2$ as

$$\rho_\omega = \rho_{t_1};\dots;\rho_{t_n}$$

End of Definition

The following definition helps us reason about events in a trace.

Definition 6.2.4 For each $t \in \Phi$ with $t = (\phi, e, \phi')$ we define the *event label* $e_t \in \mathcal{P}(D^2)$ as

$$e_t = e$$

By extension, for each, for each $\omega \in \Phi^*$ with $\omega = t_1 \dots t_n$ we define $e_\omega \in (\mathcal{P}(D^2))^*$ as

$$e_\omega = e_{t_1} \dots e_{t_n}$$

End of Definition

We introduce some short-hand notations for elements in Φ:

- (id_D, e, ϕ') can be written as (e, ϕ'),
- (ϕ, e, id_D) can be written as (ϕ, e),
- (id_D, e, id_D) can be written as e.

6.2.4 D-Systems

Now we have defined all the necessary components for our formalism, we now give a formal definition for *D-systems*.

Definition 6.2.5 Suppose we have domain $D = D_1 \times \cdots \times D_n$ consisting of n variables. We define a *D-system* as a four tuple (M, E_c, E_u, d_0), where

- M is a D-EventMachine.
- $E_c, E_u \subseteq \mathcal{P}(D^2)$ are the controllable and uncontrollable events, respectively.
- $D_0 \subseteq D$ is the set of possible initial values.

Suppose we have the following path p in M (according to Definition 6.2.2)

$$(q_0, d_0) \xrightarrow{(\phi_1, e_1, \phi_1')} (q_1, d_1) \xrightarrow{(\phi_2, e_2, \phi_2')} \cdots \xrightarrow{(\phi_n, e_n, \phi_n')} (q_n, d_n)$$

we then say p is a valid path for the D-system when $d_0 \in D_0$.

End of Definition

6.2.5 Examples

Example 6.2.2 In this example we model a simple sensor. We take $\mathbb{B}$-system

$$S = ((\{\, q_{\text{off}}, q_{\text{on}} \,\}, \{\, q_{\text{off}} \,\}, \{\, q_{\text{off}} \,\}, \Phi, \delta), \emptyset, E_u, \{\, \mathsf{false} \,\})$$

. We then have $E_u = \{\, \text{u\_on}, \text{u\_off} \,\}$ and $\Phi = \{\, (id_{\mathbb{B}}, \text{u\_on}, id_{\mathbb{B}}), (id_{\mathbb{B}}, \text{u\_off}, id_{\mathbb{B}}) \,\}$, where

$$\begin{aligned} \text{u\_on} &= \{\, (\mathsf{false}, \mathsf{true}) \,\} \\ \text{u\_off} &= \{\, (\mathsf{true}, \mathsf{false}) \,\} \end{aligned}$$

We then define δ as

$$\begin{aligned} q_{\text{off}}\text{u\_on} &= q_{\text{on}} \\ q_{\text{on}}\text{u\_off} &= q_{\text{off}} \end{aligned}$$

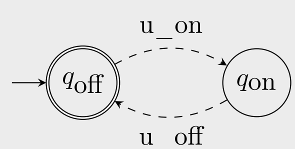

Figure 6.2.1: Visual representation of Example 6.2.2.

Example 6.2.3 In this example we model a ‘boom barrier light. We take $(\mathbb{B}\times\mathbb{B}\times\mathbb{B})$-system $S = ((\{\, q_{\text{r}}, q_{\text{rr}}, q_{\text{rg}}, q_{\text{g}} \,\}, \{\, q_{\text{r}} \,\}, \{\, q_{\text{r}} \,\}, \Phi, \delta), E_c, \emptyset, \{\, (\mathsf{false}, \mathsf{false}, \mathsf{false}) \,\})$. We then have

$$E_c = \{\, \text{c\_r}, \text{c\_rr}, \text{c\_rg}, \text{c\_g} \,\}$$

and

$$\Phi = \left\{\, (id_{(\mathbb{B}\times\mathbb{B}\times\mathbb{B})}, e, id_{(\mathbb{B}\times\mathbb{B}\times\mathbb{B})}) \mid e \in E_c \,\right\}$$

where

$$\begin{aligned} \text{c\_r} &= \{\, ((b_1, b_2, b_3), (\mathsf{true}, \mathsf{false}, \mathsf{false})) \mid b_1, b_2, b_3 \in \mathbb{B} \,\} \\ \text{c\_rr} &= \{\, ((b_1, b_2, b_3), (\mathsf{true}, \mathsf{false}, \mathsf{true})) \mid b_1, b_2, b_3 \in \mathbb{B} \,\} \\ \text{c\_g} &= \{\, ((b_1, b_2, b_3), (\mathsf{false}, \mathsf{true}, \mathsf{false})) \mid b_1, b_2, b_3 \in \mathbb{B} \,\} \\ \text{c\_rg} &= \{\, ((b_1, b_2, b_3), (\mathsf{true}, \mathsf{true}, \mathsf{false})) \mid b_1, b_2, b_3 \in \mathbb{B} \,\} \end{aligned}$$

We then define δ as

$$\begin{aligned} q_{\text{r}}\text{c\_rr} &= q_{\text{rr}} \\ q_{\text{r}}\text{c\_rg} &= q_{\text{rg}} \\ q_{\text{rr}}\text{c\_r} &= q_{\text{r}} \\ q_{\text{rg}}\text{c\_g} &= q_{\text{g}} \\ q_{\text{rg}}\text{c\_r} &= q_{\text{r}} \\ q_{\text{g}}\text{c\_r} &= q_{\text{r}} \end{aligned}$$

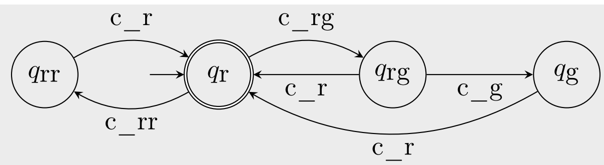

Figure 6.2.2: Visual representation of Example 6.2.3.

Example 6.2.4 In this example we model two mutual exclusive sensors. We take $(\mathbb{B} \times \mathbb{B})$-system $S = ((\{ q_{\text{off}}, q_{\text{on}} \}, \{ q_{\text{off}} \}, \{ q_{\text{off}} \}, \Phi, \delta), \emptyset, E_u, \{ (\mathsf{false}, \mathsf{false}) \})$. We then have $E_u = \{ \text{u\_on}, \text{u\_off} \}$ and $\Phi = \{ (id_{(\mathbb{B}\times\mathbb{B})}, \text{u\_on}, id_{(\mathbb{B}\times\mathbb{B})}), (id_{(\mathbb{B}\times\mathbb{B})}, \text{u\_off}, id_{(\mathbb{B}\times\mathbb{B})}) \}$, where

$$\text{u\_on} = \{ ((\mathsf{false}, \mathsf{false}), (\mathsf{true}, \mathsf{false})), ((\mathsf{false}, \mathsf{false}), (\mathsf{false}, \mathsf{true})) \}$$
$$\text{u\_off} = \{ ((\mathsf{false}, \mathsf{true}), (\mathsf{false}, \mathsf{false})), ((\mathsf{true}, \mathsf{false}), (\mathsf{false}, \mathsf{false})) \}$$

We then define δ as

$$q_{\text{off}} \text{u\_on} = q_{\text{on}}$$
$$q_{\text{on}} \text{u\_off} = q_{\text{off}}$$

A visual representation of S is given in Figure 6.2.3. A logical view of the behavior is shown in Figure 6.2.4.

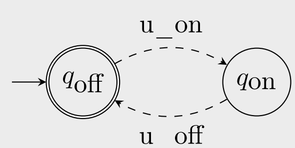

Figure 6.2.3: Visual representation of Example 6.2.4.

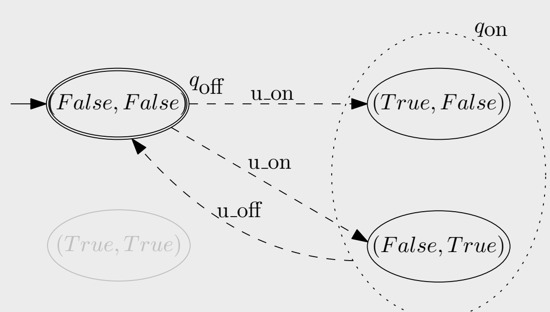

Figure 6.2.4: Representation of the behavior.

The advantages of this definition of events, opposed to the definition for EFAs, are

- Since the events are now defined as relations instead of just a label, it is easier to derive the meaning of an event from its definition. In Example 6.2.3, one can derive from definitions of the events, which lights are set on or off.
- This way of modeling events is more flexible. As we have seen in Example 6.2.4, we can model events that can have multiple possible outcomes. From an operational viewpoint, one of the outcomes is chosen non-deterministically.

To further substantiate the last point, we are going to model the input device discussed in Subsection 5.3.4. Recall that we have a set of user input commands $C = \{ c_1, \ldots, c_n \}$. We again take the domain $D = C \cup \{ \Box \}$, where $\Box$ is our 'no command' token. We now model the events 'inputCommand' and 'respond' as binary relations on D.

$$\begin{aligned} \text{inputCommand} &= \{ (\Box, c) \mid c \in C \} \\ \text{respond} &= \{ (c, \Box) \mid c \in C \} \end{aligned}$$

We then construct our D-system

$$P_i = ((\{ p_0, p_1 \}, \{ p_0 \}, \{ p_0 \}, \Phi, \delta), \{ \text{respond} \}, \{ \text{inputCommand} \}, \{ \Box \})$$

where we have $\Phi = \{ (id_D, \text{inputCommand}, id_D), (id_D, \text{respnd}, id_D) \}$, and for δ we have

$$\begin{aligned} p_0\text{inputCommand} &= p_1 \\ p_1\text{respond} &= p_0 \end{aligned}$$

A visual representation of P_i is shown in Figure 6.2.5. Observe that we use the similar method when modeling the mutual exclusive sensor in Example 6.2.4.

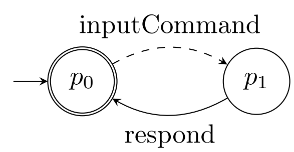

Figure 6.2.5: Visual representation of P_i

One can observe that modeling the input device as a D-system is more convenient as it was with EFA. We now only have one transition edge for the event 'inputCommand'. In Section 6.5 we discuss how we can model a requirement in this specific case.

A disadvantage of this definition is that the total state space of our model can increase. For instance, in Example 6.2.2, there is an extra Boolean value introduced which essentially coincides with the current state.

6.3 Synchronization

As with FSAs and EFAs, we define synchronous product operator for D-systems, which allows for modeling complex discrete event systems by modeling their (relatively) simple subcompontents as D-systems. These subcompontents can then be composed via synchronization (synchronous product).

Since two discrete event systems can have different domains, we need a method to compose two domains. This 'composed domain' will then be the domain of the synchronous product of the systems. Suppose we have domains D_1 and D_2. In order to compose the two domains, their *shared domain* D_s must also be given. A domain composition can then be defined as follows.

Definition 6.3.1 Given domains D_1 and D_2. Suppose there is some shared domain D_s such that $D_1 = D_1' \times D_s$ and $D_2 = D_s \times D_2'$, the domain composition ($\otimes$) is then defined as

$$D_1 \otimes D_2 = D_1' \times D_s \times D_2'$$

In the case that D_1 and D_2 are completely disjoint then the ($\otimes$) is simply defined as

$$D_1 \otimes D_2 = D_1 \times D_2$$

End of Definition

We then define an operator on binary relations on D_1 or D_2, which projects these relations on domain $D_1 \otimes D_2$.

Definition 6.3.2 Suppose we have domains D_1 and D_2 with shared domain D_2. For $\phi \in D_1^2$ we then have

$$\phi_{|D_1 \otimes D_2} = \{\, ((d_1, d_s, d_2), (d_1', d_s', d_2)) \mid (d_1, d_s, d_2) \in D_1 \otimes D_2, (d_1', d_s') \in \phi((d_1, d_s)) \,\}$$

And for $\phi \in D_2^2$ we have

$$\phi_{|D_1 \otimes D_2} = \{\, ((d_1, d_s, d_2), (d_1, d_s', d_2')) \mid (d_1, d_s, d_2) \in D_1 \otimes D_2, (d_s', d_2') \in \phi((d_s, d_2)) \,\}$$

End of Definition

We also define an operator for synchronizing a binary relation on D_1 and a binary relation on D_2.

Definition 6.3.3 Suppose we have domains D_1 and D_2 with shared domain D_s, given $\phi_1 \in D_1^2$ and $\phi_2 \in D_2^2$ we define $\phi_1 \mid\mid \phi_2$ as

$$\begin{aligned}\phi_1 \mid\mid \phi_2 = \{((d_1, d_s, d_2), (d_1', d_s', d_2')) \mid & (d_1, d_s, d_2) \in D_1 \otimes D_2, \\ & (d_1', d_s') \in \phi_1((d_1, d_s)), \\ & (d_s', d_2') \in \phi_2((d_s, d_2))\}\end{aligned}$$

End of Definition

We now define the synchronous product on two D-systems ($\mid\mid$).

Definition 6.3.4 Suppose we have D_1-system $S_1 = ((Q_1, I_1, T_1, \Phi_1, \delta_1), E_c^1, E_u^1, D_0^1)$ and D_2-system $S_2 = ((Q_2, I_2, T_2, \Phi_2, \delta_2), E_c^2, E_u^2, D_0^2)$. We then have $(D_1 \otimes D_2)$-system $S_1 \mid\mid S_2 = ((Q_1 \times Q_2, I_1 \times I_1, T_1 \times T_1, \Phi, \delta), E_c^{1\prime} \cup E_c^{2\prime}, E_u^{1\prime} \cup E_u^{2\prime}, D_0^1 \otimes D_0^2)$ where

- We first define the relevant event sets

$$E_c^{1\prime} = \left\{\, e_{|D_1 \otimes D_2} \mid e \in E_c^1 \,\right\}$$
$$E_u^{1\prime} = \left\{\, e_{|D_1 \otimes D_2} \mid e \in E_u^1 \,\right\}$$

$$E_c^{2\prime} = \left\{\, e_{|D_1 \otimes D_2} \mid e \in E_c^2 \,\right\}$$
$$E_u^{2\prime} = \left\{\, e_{|D_1 \otimes D_2} \mid e \in E_u^2 \,\right\}$$

  For brevity we also say

$$E_1' = E_c^{1\prime} \cup E_u^{1\prime}$$
$$E_2' = E_c^{2\prime} \cup E_u^{2\prime}$$

- The initial domain of the product is as follows

$$D_0^1 \otimes D_0^2 = \{ (d_1, d_s, d_2) \mid (d_1, d_s) \in D_0^1, (d_s, d_2) \in D_0^2 \}$$

- For Φ and δ we have the following.
  - For all $e \in E_1' \cap E_2'$: for each $q_1 \in Q_1$, $q_2 \in Q_2$, $(\phi_1, e, {\phi_1}') \in \Phi_1$ and $(\phi_2, e, {\phi_2}') \in \Phi_2$, $q_1(\phi_1, e, {\phi_1}') \ni q_1'$, and $q_2(\phi_2, e, {\phi_2}') \ni q_2'$, then we have $(\phi_1 \mid\mid \phi_2, e, {\phi_1}' \mid\mid {\phi_2}') \in \Phi$ and $(q_1, q_2)(\phi_1 \mid\mid \phi_2, e, {\phi_1}' \mid\mid {\phi_2}') \ni (q_1', q_2')$.
  - For all $e \in E_1' \setminus E_2'$: for each $q_1 \in Q_1$, and $(\phi_1, e, {\phi_1}') \in \Phi_1$ with $q_1(\phi_1, e, {\phi_1}') \ni q_1'$, then $(\phi_{1|D_1 \otimes D_2}, e, {\phi_1}'_{|D_1 \otimes D_2}) \in \Phi$ and for all $q_2 \in Q_2$ we have $(q_1, q_2)(\phi_{1|D_1 \otimes D_2}, e, {\phi_1}'_{|D_1 \otimes D_2}) \ni (q_1', q_2)$ in δ.
  - For all $e \in E_2' \setminus E_1'$: for each $q_2 \in Q_2$, and $(\phi_2, e, {\phi_2}') \in \Phi_2$ with $q_2(\phi_2, e, {\phi_2}') \ni q_2'$, then $(\phi_{2|D_1 \otimes D_2}, e, {\phi_2}'_{|D_1 \otimes D_2}) \in \Phi$ and for all $q_1 \in Q_1$ we have $(q_1, q_2)(\phi_{2|D_1 \otimes D_2}, e, {\phi_2}'_{|D_1 \otimes D_2}) \ni (q_1, q_2')$ in δ.

End of Definition

Example 6.3.1 Suppose we have $\mathbb{B}$-machine

$$S_1 = ((\{ q_{\text{off}}, q_{\text{on}} \}, \{ q_{\text{off}} \}, \{ q_{\text{off}} \}, \Phi_1, \delta_1), \emptyset, E_u^1, \{ \mathsf{false} \})$$

with $E_u^1 = \{ \text{u\_on}, \text{u\_off} \}$ and $\Phi = \{ (id_{\mathbb{B}}, \text{u\_on}, id_{\mathbb{B}}), (id_{\mathbb{B}}, \text{u\_on}, id_{\mathbb{B}}) \}$.

We have also $(\mathbb{B} \times \mathbb{N})$-machine

$$S_2 = ((\{ q_0, q_1 \}, \{ q_0 \}, \{ q_1 \}, \Phi_2, \delta_2), \emptyset, E_u^2, \{ (\mathsf{false}, 0) \})$$

with $E_u^2 = \left\{ \text{u\_on}_{|(\mathbb{B} \times \mathbb{N})}, \text{u\_off}_{|(\mathbb{B} \times \mathbb{N})} \right\}$ and

$$\Phi_2 = \left\{ (id_{(\mathbb{B} \times \mathbb{N})}, \text{u\_on}, id_{(\mathbb{B} \times \mathbb{N})}), (id_{(\mathbb{B} \times \mathbb{N})}, \text{u\_off}, id_{(\mathbb{B} \times \mathbb{N})}), (g_1, \text{u\_on}, u_1), (g_2, \text{u\_on}, id_{(\mathbb{B} \times \mathbb{N})}) \right\}$$

$\text{u\_on}, \text{u\_off}$ are defined as in Example 6.2.1. Furthermore we have

$$\begin{aligned}
g_1 &= \{ ((b, n), (b, n)) \mid b \in \mathbb{B}, n \in N, n < 3 \} \\
g_2 &= \{ ((b, n), (b, n)) \mid b \in \mathbb{B}, n \in N, n \geq 3 \} \\
u_1 &= \{ ((b, n), (b, n + 1)) \mid b \in \mathbb{B}, n \in N \}
\end{aligned}$$

For δ_1 we have

$$\begin{aligned}
q_{\text{off}} \text{u_on} &= q_{\text{on}} \\
q_{\text{on}} \text{u_off} &= q_{\text{off}}
\end{aligned}$$

For δ_2 we have

$$\begin{aligned}
q_0(g_1, \text{u_on}, u_1) &= q_0 \\
q_0 \text{u_off} &= q_0 \\
q_0(g_2, \text{u_on}) &= q_1 \\
q_1 \text{u_on} &= q_1 \\
q_1 \text{u_off} &= q_1
\end{aligned}$$

We then have $(\mathbb{B}, \mathbb{N})$-system

$$S_1 \mid\mid S_2 = (M, \emptyset, E_u^2, \{ (\mathsf{false}, 0) \})$$

with $(\mathbb{B}, \mathbb{N})$-EventMachine

$$M = (\{ (q_{\text{on}}, q_0), (q_{\text{off}}, q_0), (q_{\text{on}}, q_1), (q_{\text{off}}, q_1) \}, \{ (q_{\text{off}}, q_0) \}, \{ (q_{\text{off}}, q_1) \}, \delta)$$

where

$$\begin{aligned}
(q_{\text{off}}, q_0)(g_1, \text{u_on}, u_1) &= (q_{\text{on}}, q_0) \\
(q_{\text{off}}, q_0)(g_2, \text{u_on}) &= (q_{\text{on}}, q_1) \\
(q_{\text{on}}, q_0)\text{u_off} &= (q_{\text{off}}, q_0) \\
(q_{\text{off}}, q_1)\text{u_on} &= (q_{\text{on}}, q_1) \\
(q_{\text{on}}, q_1)\text{u_off} &= (q_{\text{off}}, q_1)
\end{aligned}$$

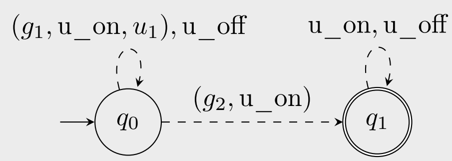

Figure 6.3.1: Visual representation of M_2

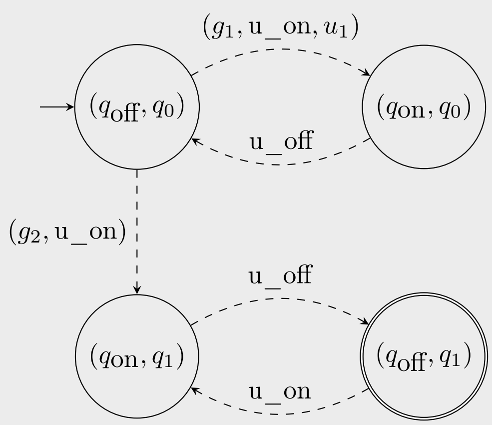

Figure 6.3.2: Visual representation of $M_1 \mid\mid M_2$

6.4 D-System Equivalence

In this section, we introduce an equivalence relation for D-systems. Such an equivalence relation can aid us in determining algebraic properties of operators and functions on D-systems (such as synchronization). To define an equivalence relation, we must determine what makes a D_1-system *equivalent* to some other D_2-system, even when the domains D_1 and D_2 are not strictly equal to each other. Since the events of D-systems are modeled after changes input/output variable values, they are the 'observable' parts of our system. For this reason, equivalent D-systems should have equivalent event traces.

In order to reason about the domain values and relations, we will first define an equivalence relation on said domain values and relations.

Definition 6.4.1 Suppose we have domains D_1 and D_2. Suppose we have $d \in (D_1 \otimes D_2)$ with $d = (d_1, d_s, d_2)$ and $d' \in (D_2 \otimes D_1)$ with $d' = (d'_2, d'_s, d'_1)$. We define $\equiv$ as

$$d \equiv d' \iff (d_1 = d'_1) \wedge (d_s = d'_s) \wedge (d_2 = d'_2)$$

Suppose we have $d_1 \in D_1$ with $d_1 = (d'_1, d^1_s)$, $d_2 \in D_2$ with $d_2 = (d'_2, d^2_s)$, and $d \in D_1 \otimes D_2$ with $d = (d''_1, d_s, d''_2)$. We then say

$$\begin{aligned} d_1 \equiv d &\iff d'_1 = d''_1 \wedge d^1_s = d_s \\ d_2 \equiv d &\iff d'_2 = d''_2 \wedge d^2_s = d_s \end{aligned}$$

End of Definition

According to the equivalence relations defined in Definition 6.4.1, it does not matter in which order the domains are composed. Based on this equivalence relation we define the following ordering on relations on composed domains.

Definition 6.4.2 Suppose we have $\phi \subseteq (D_1 \otimes D_2)^2$ and $\phi' \subseteq (D_2 \otimes D_1)^2$, then we have

$$\begin{aligned} \phi \sqsubseteq \phi' &\iff (\forall (d_1, d_2) \in \phi : \exists (d'_1, d'_2) \in \phi' : d_1 \equiv d'_1 \wedge d_2 \equiv d'_2) \\ \phi' \sqsubseteq \phi &\iff (\forall (d'_1, d'_2) \in \phi' : \exists (d_1, d_2) \in \phi : d'_1 \equiv d_1 \wedge d'_2 \equiv d_2) \end{aligned}$$

End of Definition

From which the following definition follows:

Definition 6.4.3 Suppose we have $\phi \in (D_1 \otimes D_2)$ and $\phi' \in (D_2 \otimes D_1)$, then we have

$$\phi \equiv \phi' \iff \phi \sqsubseteq \phi' \wedge \phi' \sqsubseteq \phi$$

End of Definition

Now we can define an ordering ($\sqsubseteq$) on the traces of tuples (ϕ, e, ϕ') used in EventMachines. This ordering can both be used on two labels (traces) on the same domain D and on a label on $D_1 \otimes D_2$ and a label of $D_2 \otimes D_1$ (where we need to use the ordering defined in Definition 6.4.2). We use the ρ_t from Definition 6.2.3 to compare the relations of the label. Since we want equivalent event labels in equivalent systems, we only compare traces when their respective event labels are equivalent (for which we use e_t from Definition 6.2.4).

Definition 6.4.4 Suppose we have event set E on domain D and $\Phi \subseteq (\mathcal{P}(D^2) \times E \times \mathcal{P}(D^2))$. Then we define ($\sqsubseteq$) on $\omega, \omega' \in \Phi^*$ as

$$\omega \sqsubseteq \omega' \iff \rho_\omega \subseteq \rho'_\omega \wedge e_\omega = e_{\omega'}$$

Suppose we have event set E on domain $D_1 \otimes D_2$, event set E' on domain $D_2 \otimes D_1$, $\Phi \subseteq \mathcal{P}((D_1 \otimes D_2)^2) \times E \times \mathcal{P}((D_1 \otimes D_1)^2)$ and $\Phi' \subseteq \mathcal{P}((D_2 \otimes D_1)^2) \times E' \times \mathcal{P}((D_2 \otimes D_1)^2)$. In this case the definition of ($\sqsubseteq$) on $\omega \in \Phi^*$ and $\omega' \in \Phi'^*$ is as follows.

$$\begin{aligned} \omega \sqsubseteq \omega' &\iff \rho_\omega \sqsubseteq \rho_{\omega'} \wedge e_\omega \equiv e_{\omega'} \\ \omega' \sqsubseteq \omega &\iff \rho_{\omega'} \sqsubseteq \rho_\omega \wedge e_{\omega'} \equiv e_\omega \end{aligned}$$

For both cases, we define ($\equiv$) on ω and ω' as

$$\omega \equiv \omega' \iff \omega \sqsubseteq \omega' \wedge \omega' \sqsubseteq \omega$$

End of Definition

For D-systems with equivalent event traces, we can now define an ordering ($\preccurlyeq$). The intuition of $S_1 \preccurlyeq S_2$ is that S_2 has 'at least' the behavior of S_1. This means that for every path with label ω in S_1, there is a path in S_2 with label ω' where $\omega \sqsubseteq \omega'$ (meaning that the paths have equivalent event labels, but the trace from S_2 may have a 'bigger' label relation).

Definition 6.4.5 Suppose we have D_1-system $S_1 = ((Q_1, I_1, T_1, \Phi_1, \delta_1), E_c^1, E_u^1, D_0^1)$ and D_2-system $S_2 = ((Q_2, I_2, T_2), E_c^2, E_u^2, D_0^2)$. We define ordering ($\preccurlyeq$) on systems as:

$S_1 \preccurlyeq S_2$ if and only if

- Suppose there is the path
$$q_0^1 \xrightarrow{t_1^1} q_1^1 \xrightarrow{t_2^1} q_2^1 \xrightarrow{t_3^1} \dots \xrightarrow{t_n^1} q_n^1$$
in S_1 with label $\omega^1 = t_1^1 \dots t_n^1$, then there exists path
$$q_0^2 \xrightarrow{t_1^2} q_1^2 \xrightarrow{t_2^2} q_2^2 \xrightarrow{t_3^2} \dots \xrightarrow{t_m^2} q_m^2$$
in S_2 with label $\omega^2 = t_1^2 \dots t_m^2$, such that $\omega^1 \sqsubseteq \omega^2$.

End of Definition

From the ordering ($\preccurlyeq$) we define our equivalence relation ($\equiv$) on D-systems.

Definition 6.4.6 Suppose we have D_1-system $S_1 = ((Q_1, I_1, T_1, \Phi_1, \delta_1), E_c^1, E_u^1, D_0^1)$ and D_2-system $S_2 = ((Q_2, I_2, T_2), E_c^2, E_u^2, D_0^2)$. We define the equivalence ($\backsimeq$) on systems as follows

$$S_1 \backsimeq S_2 \iff S_1 \preccurlyeq S_2 \wedge S_2 \preccurlyeq S_1$$

End of Definition

In Appendix A proof outlines are given for associativity and commutativity properties of the parallel composition operator $||$ under the equivalence relation $\equiv$. If an operator has these two properties, then the order in which operands are applied to the operator does not matter.

6.5 D-System Based Requirements

We now discuss how we can use requirements modeled by a D-System R to refine a plant modeled by a D-system P, as previously discussed with EFAs.

Definition 6.5.1 Suppose our plant is modeled by D_P-system P, and our requirements are modeled by D_R-system R. Using synchronization, the *refined plant* is defined as

$$P_R = P \,||\, R$$

End of Definition

Example 6.5.1 Suppose we model a simple sensor $\mathbb{B}$-system

$$S = ((\{ q_{\text{off}}, q_{\text{on}} \}, \{ q_{\text{off}} \}, \{ q_{\text{off}} \}, \Phi, \delta), \emptyset, E_u, \{ \mathsf{false} \})$$

as shown in Figure 6.5.1. We then have $E_u = \{ \text{s\_switch} \}$ and $\Phi = \{ (id_{\mathbb{B}}, \text{s\_switch}, id_{\mathbb{B}}) \}$, where

$$\text{s\_switch} = \{ (\mathsf{false}, \mathsf{true}), (\mathsf{true}, \mathsf{false}) \}$$

We then define δ as

$$q_{\text{off}} \text{s\_switch} = q_{\text{on}}$$
$$q_{\text{on}} \text{s\_switch} = q_{\text{off}}$$

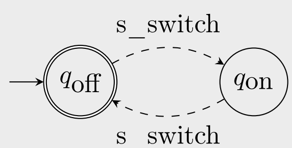

Figure 6.5.1: A simple sensor

We also model a simple actuator $\mathbb{B}$-system

$$A = ((\{ q_{\text{off}}, q_{\text{on}} \}, \{ q_{\text{off}} \}, \{ q_{\text{off}} \}, \Phi, \delta), E_c, \emptyset, \{ \mathsf{false} \})$$

as shown in Figure 6.5.2. We then have $E_c = \{ \text{a\_switch} \}$ and $\Phi = \{ (id_{\mathbb{B}}, \text{a\_switch}, id_{\mathbb{B}}) \}$, where

$$\text{a\_switch} = \{ (\mathsf{false}, \mathsf{true}), (\mathsf{true}, \mathsf{false}) \}$$

We then define δ as

$$q_{\text{off}} \text{a\_switch} = q_{\text{on}}$$
$$q_{\text{on}} \text{a\_switch} = q_{\text{off}}$$

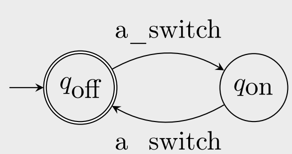

Figure 6.5.2: A simple actuator.

We then have our plant $(\mathbb{B} \times \mathbb{B})$-system $P = S \,||\, A$ (Figure 6.5.3).

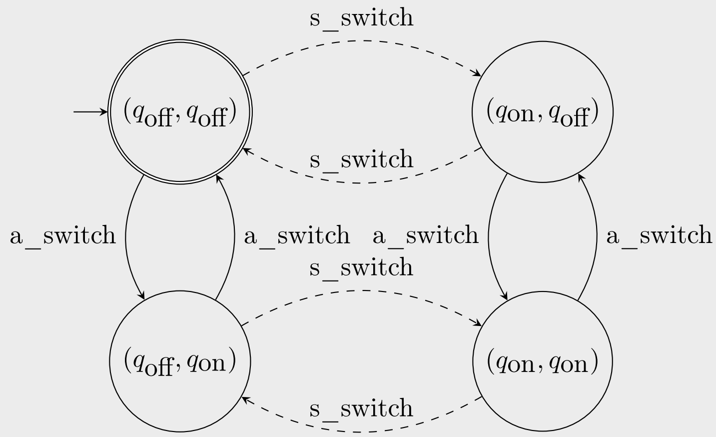

Figure 6.5.3: Visual representation of $P = S \,||\, A$

We now model the follow requirement:
'The actuator may only switch on/off, after the sensor has switched on/off.'
Essentially, we say that the event a_switch may only occur after an occurrence of event s_switch. The requirement is modeled by $(\mathbb{B}_s \times \mathbb{B}_a)$-system

$$R = ((\{ r_0, r_1 \}, \{ r_0 \}, \{ r_0 \}, \Phi, \delta), E_c, E_u, \{ (\mathsf{false}, \mathsf{false}) \})$$

as shown in Figure 6.5.4 with

$$\Phi = \{ (id_{(\mathbb{B}_s \times \mathbb{B}_a)}, \text{s\_switch}, id_{(\mathbb{B}_s \times \mathbb{B}_a)}), (id_{(\mathbb{B}_s \times \mathbb{B}_a)}, \text{a\_switch}, id_{(\mathbb{B}_s \times \mathbb{B}_a)}) \}$$

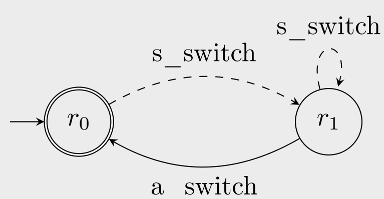

Figure 6.5.4: Visual representation of R.

Example 6.5.2 Given the sensor and actuator as given in Example 6.5.1, we model the following requirement:
'The actuator may only switch on/off, after the sensor has been activated.'
The requirement is modeled by $(\mathbb{B}_s, \mathbb{B}_a)$-system R (Figure 6.5.5), with type
$\Phi = \{ (\text{s\_switch}, i_{off}), (\text{s\_switch}, i_{on}), \text{s\_switch}, \text{a\_switch} \}$ where

$$i_{off} = \{ ((\mathsf{false}, b_a), (\mathsf{false}, b_a)) \mid b_a \in \mathbb{B} \}$$
$$i_{on} = \{ ((\mathsf{true}, b_a), (\mathsf{true}, b_a)) \mid b_a \in \mathbb{B} \}$$

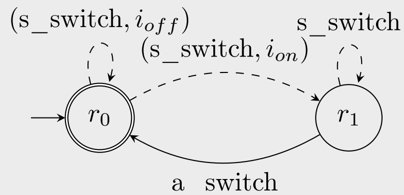

Figure 6.5.5: Visual representation of R

Example 6.5.3 Given the sensor and actuator as given in Example 6.5.1, we model the following requirement:
'The actuator may only switch on/off, after sensor has been activated 3 times.'
The requirement is modeled by $(\mathbb{B}_s, \mathbb{B}_a, \mathbb{N})$-system R (Figure 6.5.6) where

$$
\begin{aligned}
g_1 &= \{\, ((b,b,n),(b,b,n)) \mid b,b \in \mathbb{B}, n < 3 \,\} \\
g_2 &= \{\, ((b,b,n),(b,b,n)) \mid b,b \in \mathbb{B}, n \geq 3 \,\} \\
u_1 &= \{\, ((b,b,n),(b,b,n+1)) \mid b,b \in \mathbb{B}, n \in \mathbb{N} \,\} \\
u_2 &= \{\, ((b,b,n),(b,b,0)) \mid b,b \in \mathbb{B}, n \in \mathbb{N} \,\} \\
i_{off} &= \{\, ((\mathsf{false}, b_a),(\mathsf{false}, b_a)) \mid b_a \in \mathbb{B} \,\} \\
i_{on} &= \{\, ((\mathsf{true}, b_a),(\mathsf{true}, b_a)) \mid b_a \in \mathbb{B} \,\}
\end{aligned}
$$

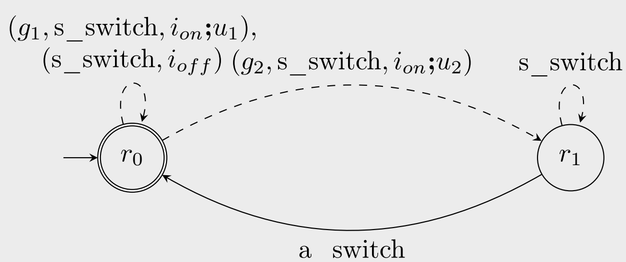

Figure 6.5.6: Visual representation of R

Example 6.5.4 Suppose we have a mutex sensor modeled by $(\mathbb{B} \times \mathbb{B})$-system

$$S = ((\{\, q_{\text{off}}, q_{\text{on}} \,\}, \{\, q_{\text{off}} \,\}, \{\, q_{\text{off}} \,\}, \Phi, \delta), \emptyset, E_c, \{\, (\mathsf{false}, \mathsf{false}) \,\})$$

as shown in Figure 6.5.7, where $E_u = \{\, \text{s\_switch} \,\}$ and $\Phi = \left\{\, (id_{(\mathbb{B}\times\mathbb{B})}, \text{s\_switch}, id_{(\mathbb{B}\times\mathbb{B})}) \,\right\}$, where

$$
\begin{aligned}
\text{s\_switch} = \{&((\mathsf{false},\mathsf{false}),(\mathsf{true},\mathsf{false})),((\mathsf{false},\mathsf{false}),(\mathsf{false},\mathsf{true})) \\
&((\mathsf{false},\mathsf{true}),(\mathsf{false},\mathsf{false})),((\mathsf{true},\mathsf{false}),(\mathsf{false},\mathsf{false}))\}
\end{aligned}
$$

where δ is defined as

$$
\begin{aligned}
q_{\text{off}}\text{s\_switch} &= q_{\text{on}} \\
q_{\text{on}}\text{s\_switch} &= q_{\text{off}}
\end{aligned}
$$

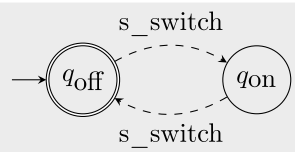

Figure 6.5.7: Visual representation of S.

Given the mutex sensor and the actuator as given in Example 6.5.1, we model the following requirement:
'The actuator may only switch after alternating sensor activation.'
The requirement is modeled by $(\mathbb{B}_{s_1}, \mathbb{B}_{s_2}\mathbb{B}_a, \mathbb{B})$-system R (Figure 6.5.8), where

$$\begin{aligned}
i_1 &= \{\, ((\mathsf{false}, b_{s_2}, b_a, \mathsf{false}), (\mathsf{false}, b_{s_2}, b_a, \mathsf{false})) \mid b_{s_2} \in \mathbb{B}_{s_2}, b_a \in \mathbb{B}_a \,\} \\
&\quad \cup \{\, ((b_{s_1}, \mathsf{false}, b_a, \mathsf{true}), (b_{s_1}, \mathsf{false}, b_a, \mathsf{true})) \mid b_{s_1} \in \mathbb{B}_{s_1}, b_a \in \mathbb{B}_a \,\} \\
i_2 &= \{\, ((\mathsf{true}, b_{s_2}, b_a, \mathsf{false}), (\mathsf{true}, b_{s_2}, b_a, \mathsf{true})) \mid b_{s_2} \in \mathbb{B}_{s_2}, b_a \in \mathbb{B}_a \,\} \\
&\quad \cup \{\, ((b_{s_1}, \mathsf{true}, b_a, \mathsf{true}), (b_{s_1}, \mathsf{true}, b_a, \mathsf{false})) \mid b_{s_1} \in \mathbb{B}_{s_1}, b_a \in \mathbb{B}_a \,\}
\end{aligned}$$

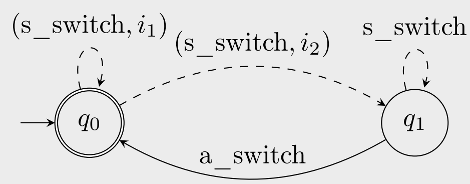

Figure 6.5.8: Visual representation of R.

Recall the D-system model of the input device from Subsection 5.3.4, introduced in Subsection 6.2.5. We now discuss how a requirement concerning the input device can be modeled as a D-system. Again we want to model the desired behavior after the system receives some command c_k from the user. R_j is then modeled by the $(D \times D')$-system

$$R_j = ((\{\, r_0, r_1, \dots, r_m \,\}, \{\, r_0 \,\}, \{\, r_0 \,\}, \Phi, \delta), \{\, \text{respond}', \dots \,\}, \{\, \text{inputCommand}', \dots \,\}, \{\, \Box \,\})$$

Again, we have $D = C \cup \{\, \Box \,\}$ and D' are other variables (if needed). inputCommand$'$ and respond$'$ are the respective events inputCommand and respond projected on the domain $D \times D'$. We then have

$$\begin{aligned}
\Phi = \{&(id_{(D \times D')}, \text{respond'}, id_{(D \times D')}), \\
&(id_{(D \times D')}, \text{inputCommand}', \text{otherCommand}), \\
&(id_{(D \times D')}, \text{inputCommand}', \text{checkCommand}), \\
&\dots \\
&\}
\end{aligned}$$

Where we have the relations 'checkCommand', which checks if the given command is c_k, and 'otherEvent', which checks if the given command is not c_k. These relations are defined as follows.

$$\begin{aligned}
\text{checkCommand} &= \{\, ((c_k, d'), (c_k, d')) \mid d' \in D' \,\} \\
\text{otherCommand} &= \{\, ((c, d'), (c, d')) \mid d' \in D', c \in C \setminus \{\, c_k \,\} \,\}
\end{aligned}$$

For δ we have

$$\begin{aligned}
r_0(\text{inputCommand}', \text{otherCommand}) &= r_0 \\
r_0\text{respond}' &= r_0 \\
r_0(\text{inputCommand}', \text{checkCommand}) &= r_1 \\
&\dots \\
r_m\text{respond}' &= r_0
\end{aligned}$$

A visual representation of R_j is shown in Figure 6.5.9.

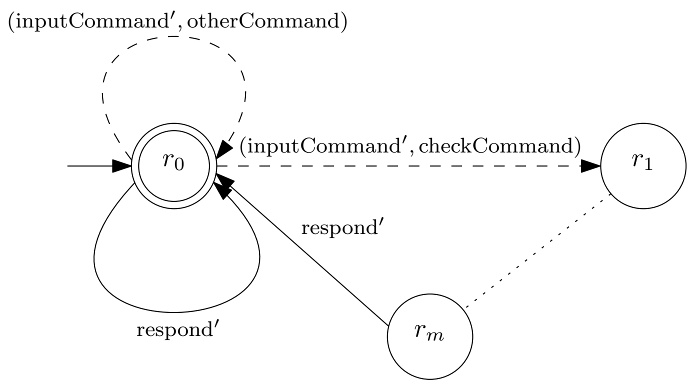

Figure 6.5.9: The requirement R_j

We can observe that modeling requirements with this approach is more convenient than the first proposed approach using EFAs in Subsection 5.3.4.

6.6 Restrictions

As explained in the second discussed issue of Subsection 5.3.4, separate subcomponents of a plant can have inter-dependencies. An example of this would be a sensor which can only activate when some actuator is activated in practical situations. In order to address this we introduce *restrictions*, which can be used to prevent certain events from happening in certain states of a plant.

Definition 6.6.1 Suppose we have D-system $S = ((Q, I, T, \Phi, \delta), E_c, E_u, D_0)$. We define a *restriction* as a tuple of type $(E_c \cup E_u) \times Q$. For a restriction (e, q) we then say that q *restricts event* e.

End of Definition

We then define the operation $(\downarrow)$ which 'applies' a set of these restrictions to some plant D-system. The operation $(\downarrow)$ can then be used to deal with the physical restrictions as discussed in Subsection 5.3.4.

Definition 6.6.2 Suppose we have D-system $S = ((Q, I, T, \Phi, \delta), E_c, E_u, D_0)$. We define the *restrict* operator $(\downarrow)$. Given S and a set of restrictions $R \subseteq \mathcal{P}((E_c \cup E_u) \times Q)$, $(\downarrow)$ will return a D-system S' which is S with the restrictions from R applied:

$$S \downarrow R = ((Q, I, T, \Phi, \delta'), E_c, E_u, D_0)$$

with

$$q(\phi, e, \phi') \ni q' \text{ in } \delta' \iff q(\phi, e, \phi') \ni q' \text{ in } \delta \wedge (e, q) \notin R$$

End of Definition

Example 6.6.1 Suppose we have $P = S \,||\, A$ as given in Example 6.5.1, and that we have the following set of restrictions.

$$R = \{\, (\text{s\_switch}, (q_{\text{off}}, q_{\text{off}})), (\text{s\_switch}, (q_{\text{on}}, q_{\text{on}})) \,\}$$

$P \downarrow R$ is shown in Figure 6.6.1, where transitions which are in P, but are not in $P \downarrow R$ are drawn in red.

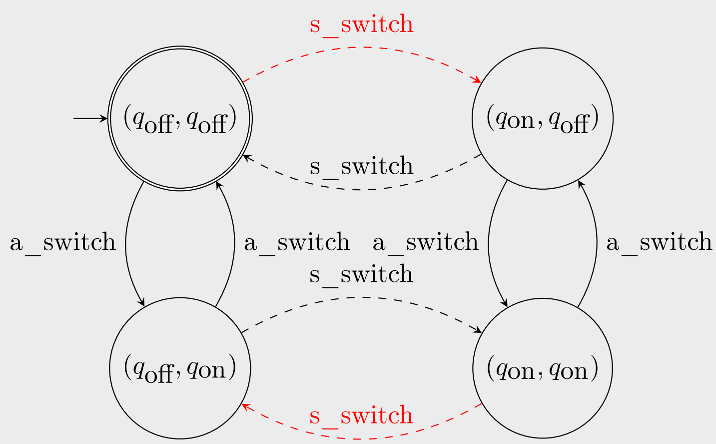

Figure 6.6.1: Visual representation of $P \downarrow R$.

6.7 Supervisory Control

Suppose we have our plant D-system $P = ((Q, I, T, \Phi, \delta), E_c, E_u, D_0)$. Let P_R be our refined plant with respect to some requirements D-system R. We now introduce the notions of non-blockingness, controllability and (proper) supervisor for D-systems.

- $q \in Q$ is *non-blocking* if for each $d \in D$ there exists $t \in \Phi^*$ such that $qt \cap T \neq \emptyset$ and for the label relation we have $\rho_t(d) \neq \emptyset$.
- P is *non-blocking* if each reachable state in P is non-blocking.
- Suppose we have D'-system S. S is *controllable* with respect to P when: if there is some $q \in Q$, $d \in D$ and $e \in E_u$ with some transition $q\phi \neq \emptyset$ and $(\phi;e;\phi')(d) \neq \emptyset$, then for every (q, q') in $P \,||\, S$ and $(d, d') \in D \otimes D'$ there exists some transition $(q, q')(\phi'', e_{|D\otimes D'}, \phi''') \neq \emptyset$ in $P \,||\, S$ for which $(\phi'';e_{|D\otimes D'};\phi''')((d, d')) \neq \emptyset$.
- $D \times D$-machine S is a *proper supervisor* for P if $S \preccurlyeq P_R$, S is controllable with respect to P and $P \,||\, S$ is non-blocking.
- Suppose we have proper supervisor S. S is a *maximally permissive supervisor* if for any proper supervisor S' we have $S' \preccurlyeq S$.

6.8 Supervisory Control Synthesis

In this section, we discuss a solution for the supervisory control problem, meaning that we define an algorithm for computing a maximally permissive supervisor for (refined) plant P. In order to make the problem easier to solve, we first present a reduction of the problem domain to a more concise domain. We then present an algorithm for this reduced version of the problem. Lastly, we present the complete algorithm, which combines the reduction procedure with the aforementioned algorithm.

6.8.1 Intuition of the Reduction

As discussed in Subsection 5.3.1, an EFA can be reduced to an FSA, which essentially comes down to eliminating the domain D by reducing it to extra state space. We can also eliminate the state space L by reducing it to an extra domain variable, meaning that we end up with a new domain $D' = D \times L$.

Suppose we have EFA $G = (L, D, \Sigma, E, L_0, D_0, L_m)$. Our simplified EFA is defined as

$$G' = (\{ l_0, l_1 \}, D \times L, \Sigma \cup \{ \sigma_f \}, E', \{ l_0 \}, D_0 \times L_0, \{ l_1 \})$$

with

- For each $e \in E$, we have $e' \in E'$ such that
  - $o'_e = l_0$,
  - $t'_e = l_0$,
  - $\sigma'_e = \sigma_e$,
  - $g'_e((d, l)) = g_e(d) \wedge l = o_e$,
  - $f'_e((d, l)) = (f_e(d), t_e)$.
- And we have $e' \in E$ such that
  - $o'_e = l_0$,
  - $t'_e = l_1$,
  - $\sigma'_e = \sigma_f$,
  - $g'_e((d, l)) = l \in L_m$,
  - $f'_e((d, l)) = id_{D \times L}$.

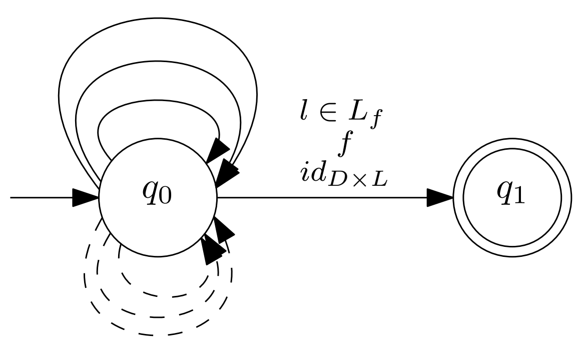

Figure 6.8.1: The simplified plant

Based on this concept we can reduce the EFA G to two sets of 'updates' on a domain $D' = D \times L$ of the form (g, u, σ), where g is the guard predicate, u is the update function of type $D' \to D'$ and σ is the event. The updates with $\sigma \in \Sigma_c$ are in the set of controllable updates C, and the updates with $\sigma \in \Sigma_u$ are in the set of uncontrollable updates U. We also introduce a finalization predicate $F : D' \to \mathbb{B}$, with $F((d, l)) = l \in L_m$.

We can now apply the same concept to D-systems.

Suppose we have plant D-system $P = ((Q, I, T, \Phi, \delta), E_c, E_u, D_0)$. We can construct $(D \times Q)$-system $S' = ((\{ q_i, q_f \}, \{ q_i \}, \{ q_f \}, \Phi', \delta'), E_c, E_u, D_0)$. For every $(\phi, e, \phi') \in \Phi$ then for all $q \in Q$ with $q(\phi, e, \phi') = Q' \neq \emptyset$ we have

$$t = (\phi_{|D\times Q}, e_{|D\times Q}, \phi'_{|D\times Q}; \{ ((d, q), (d, q')) \mid d \in D, q' \in Q' \})$$

with $t \in \Phi'$ and $q_i t = q_i$ in δ'. Furthermore, we have $f = (\{ ((d, q), (d, q)) \mid d \in D, q \in T \}, \tau, id_{D\times Q}) \in \Phi'$ and $q_i f = q_f$. A visual representation of P' is shown in Figure 6.8.2.

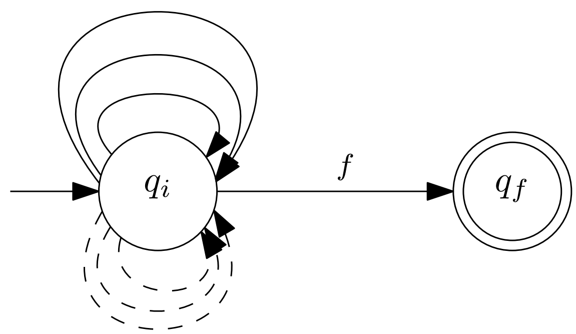

Figure 6.8.2: The reduced plant

Observe that we now essentially have a set of controllable and uncontrollable updates $C, U \subseteq \Phi'$, and a finalization predicate $F((d, q)) = q \in T$.

Note In order to reason that the reduced plant is equivalent to the original plant, we have to change the definition of $(\simeq)$, since the current definition does not allow that the two D-systems have different domains.

6.8.2 Reduction of the Problem Domain

Based on the intuition discussed in the previous section, we now define our reduction procedure on the supervisory control problem.

Suppose we we have plant refined plant D-system

$$P = ((Q, I, T, \Phi, \delta), E_c, E_u, D_0)$$

, we compute the update sets C and U, and the finalization predicate F as follows:

1. Let $D' = D \times Q$
2. Let C and U be two empty sets.
3. For each $q(\phi, e, \phi') = Q'$ in δ:
    - Let $t = (\phi_{|D\times Q}, e_{|D\times Q}, \phi'_{|D\times Q}; \{ ((d, q), (d, q')) \mid d \in D, q' \in Q' \})$.

   - Add t to C if e is controllable, otherwise add t to U.

4. Let $F : D' \to \mathbb{B}$ such that $N_0(d, q) = q \in T$.

We will now define the relevant definitions for supervisory control synthesis on domain D, sets $C, U \subseteq \mathcal{P}(D^2)^3$ and finalization predicate $F : D \to \mathbb{B}$.

- We define the ordering $\preceq$ as $(C', U') \preceq (C, U)$ if for each $t' \in (C' \cup U')$ with $t' = (\phi', e, \phi'')$ there is $t \in (C \cup U)$ with $t = (\phi, e, \phi')$ for which each $(d, d') \in \rho_{t'}$ we have $(d, d') \in \rho_t$.
- $d \in D$ is *non-blocking* with respect to C and U if there exists $\omega \in (C \cup U)^*$ such that $d' \in \rho_\omega(d)$ and $F(d') = \mathsf{true}$.
- $d \in D$ is *completely non-blocking* with respect to C and U if there exists $\omega \in (C \cup U)^*$ such that $d' \in \rho_\omega(d)$ is blocking.
- C and U are non-blocking if for each non-blocking $d \in D$, d is either completely non-blocking or if there exists $\omega \in (C \cup U)^*$ for which there is $d' \in \rho_\omega(d)$ which is blocking, then $\omega \in U^*$.
- Suppose we have $C', U' \subseteq \mathcal{P}(D^2)^3$. C' and U' are *controllable* with respect to C and U if for each $t \in U$ with $t = (\phi, e, \phi')$ for which $d \in D$ has $\rho_t(d) \neq \emptyset$, there is $t' \in U'$ with $t' = (\phi'', e, \phi''')$ for which $\rho_{t'}(d) \neq \emptyset$.
- C' and U' form a *proper supervisor* for C and U if $(C', U') \preceq (C, U)$, C' and U' are controllable with respect to C and U, and C' and U' are non-blocking.
- C' and U' form a *maximally permissive proper supervisor* for C and U when, for any other proper supervisor C'' and U'', we have the following. Suppose we have $d \in D$ which is completely non-blocking with respect to C'' and U''. Let $\omega' \in (C'' \cup U'')$ with $d' \in \rho_{\omega'}(d)$, then there exists $\omega \in (C' \cup U')^*$ such that $d' \in \rho_\omega(d)$.

The supervisory control problem is now defined as follows: Given domain D, controllable and uncontrollable updates C and U, and finalization predicate $F : D \to \mathbb{B}$, compute a maximally permissive supervisor for C and U.

Suppose we have a maximally permissive supervisor for C and U, formed by C' and U', then we can create a maximally permissive supervisor S for our refined plant P as follows: construct supervisor D'-system $S = ((\{q\}, \{q\}, \{q\}, C' \cup U', \delta'), E_c, E_u, D_0 \times I)$ with for all $t \in C' \cup U'$ we have $qt = q$.

6.8.3 Algorithm for Simplified Problem

Given domain D, controllable and uncontrollable updates C and U and finalization predicate $F : D \to \mathbb{B}$, Algorithm 3 computes controllable updates set C' such that C' and U are the most permissive supervisor for C and U.

Algorithm 3: Supervisory Synthesis for D-systems

Data: Domain D, Set of controllable and uncontrollable updates C and U, finalization predicate $B : D \to \mathbb{B}$

Result: Update relations C

$i \leftarrow 0$;
$C_0 \leftarrow C$;
$\forall d \in D : B(d) \leftarrow \mathsf{false}$;
do
  $N \leftarrow$ `FixPredicate(`B, $C_i \cup U$`)` ;
  $\forall d \in D : B(d) \leftarrow \neg N(d) \vee B(d)$;
  $B \leftarrow$ `FixPredicate(`B,U`)` ;
  $\forall(\phi, e, \phi') \in C_i : (\phi, e, \phi' \mathbf{;} \{ (d, d) \mid d \in D, \neg B(d) \}) \in C_{i+1}$;
  $i \leftarrow i + 1$;
while $C_{i-1} \neq C_i$;
return C_{i-1}

Function `FixPredicate(`$P : D \to \mathbb{B}$, $R \subseteq \mathcal{P}(D^2)$`)`
  $i \leftarrow 0$;
  $P_0 \leftarrow P$;
  do
    $P_{i+1}(d) \leftarrow P_i(d) \vee \bigvee_{\phi \in R} \bigvee_{d' \in \rho_\phi(d)} P_i(d')$;
    $i \leftarrow i + 1$;
  while $\neg \forall d \in D : P_{i-1}(d) = P_i(d)$;
  return P_{i-1}

Observe that this algorithm is mostly based on supervisory control synthesis algorithm for EFAs (Algorithm 2). In Appendix B, an outline of the proof of correctness for Algorithm 3. The main purpose of creating this outline, is to give clarity in what properties the algorithm adheres to, and where the definitions of Subsection 6.8.2 should be adjusted.

6.8.4 Complete Algorithm

We compute a proper supervisor for some refined plant D-system $P = ((Q, I, T, \Phi, \delta), E_c, E_u, D_0)$ as follows

1. Let $D' = D \times Q$
2. Let C and U be two empty sets.
3. For each $q(\phi, e, \phi') = Q'$ in δ:
    - Let $t = (\phi_{|D \times Q}, e_{|D \times Q}, \phi'_{|D \times Q} \mathbf{;} \{ ((d, q), (d, q')) \mid d \in D, q' \in Q' \})$.
    - Add t to C if e is controllable, otherwise add t to U.
4. Let $F : D' \to \mathbb{B}$ such that $N_0(d, q) = q \in T$.
5. Use Algorithm 3 with input (D', C, U, N_0) to compute C'.
6. Construct supervisor D'-system $S = ((\{ q \}, \{ q \}, \{ q \}, C' \cup U, \delta'), E_c, E_u, D_0 \times I)$ with for all $t \in C' \cup U$ we have $qt = q$.

Observe that the result D'-machine S is a maximally permissive supervisor for P.

Example 6.8.1 We base this example on Example 5.3.3. Suppose we have our (refined) plant $(\mathbb{B} \times \mathbb{B} \times \{0, \ldots, 10\})$-system $P = ((\{q_0, q_1, q_2\}, \{q_0\}, \{q_0\}, \Phi, \delta), E_c, E_u, D_0)$ (Figure 6.8.3). $E_c = \{\text{s\_switch}\}$, $E_u = \{\text{a\_switch}\}$ and

$$\Phi = \{(g_1, \text{a\_switch}, u), (g_2, \text{a\_switch}, u), (g_3, \text{s\_switch}, u)\}$$

with

$$\begin{aligned}
\text{s_switch} &= \{((b_s, b_a, n), (\neg b_s, b_a, n))\} \\
\text{a_switch} &= \{((b_s, b_a, n), (b_s, \neg b_a, n))\}
\end{aligned}$$

$$\begin{aligned}
g_1 &= \{((b_s, b_a, n), (b_s, b_a, n)) \mid n < 8\} \\
g_2 &= \{((b_s, b_a, n), (b_s, b_a, n)) \mid n < 9\} \\
g_3 &= \{((b_s, b_a, n), (b_s, b_a, n)) \mid b_s, b_a \in \mathbb{B}, n \in \mathbb{N}, n < 7\} \\
u &= \{((b_s, b_a, n), (b_s, b_a, n+1)) \mid b_s, b_a \in \mathbb{B}, n \in \mathbb{N}\}
\end{aligned}$$

δ is defined as

$$\begin{aligned}
q_0(g_1, \text{a_switch}, u) &= q_1 \\
q_1(g_2, \text{a_switch}, u) &= q_2 \\
q_2(g_3, \text{s_switch}, u) &= q_0
\end{aligned}$$

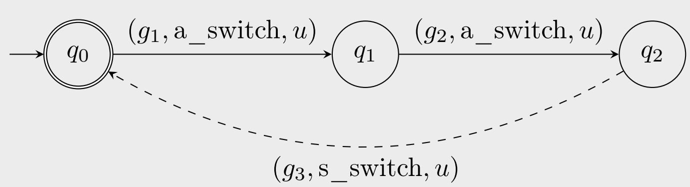

Figure 6.8.3: Visual representation of S.

We now compute the most-permissive supervisor for S.

- We first construct the sets C and U of binary relations on $(\mathbb{B} \times \mathbb{B} \times \{0, \ldots 10\} \times \{q_0, q_1, q_2\})$.

$$\begin{aligned}
U &= \{(g_3, \text{s_switch}, u; \{((b_s, b_a, n, q_2), (b_s, b_a, n, q_0))\})\} \\
C &= \{(g_1, \text{a_switch}, u; \{((b_s, b_a, n, q_0), (b_s, b_a, n, q_1))\}) \\
&\quad (g_2, \text{a_switch}, u; \{((b_s, b_a, n, q_1), (b_s, b_a, n, q_2))\})
\end{aligned}$$

  We also construct the finalization predicate $F(((b_s, b_a, n, q)) = q \in \{q_0\}$.

- We perform the first iteration of the algorithm. First the non-blocking predicate N is computed:

$$
\begin{aligned}
N((b_s, b_a, n, q_0)) &= \mathsf{true} \\
N((b_s, b_a, n, q_2)) &= \mathsf{true} \iff n < 7 \\
N((b_s, b_a, n, q_1)) &= \mathsf{true} \iff n < 6
\end{aligned}
$$

Then the bad predicate B is computed

$$
\begin{aligned}
B((b_s, b_a, n, q_0)) &= \mathsf{false} \\
B((b_s, b_a, n, q_1)) &= \mathsf{true} \iff n \geq 6 \\
B((b_s, b_a, n, q_2)) &= \mathsf{true} \iff n = 7
\end{aligned}
$$

We then update the controllable update relations:

$$
\begin{aligned}
C_1 = \{&(g_1, \text{a\_switch}, u; \{\, ((b_s, b_a, n, q_0), (b_s, b_a, n, q_1)) \,\}; \{\, ((b_s, b_a, n, q), (b_s, b_a, n, q)) \mid n < 6 \,\}) \\
&(g_2, \text{a\_switch}, u; \{\, ((b_s, b_a, n, q_1), (b_s, b_a, n, q_2)) \,\}; \{\, ((b_s, b_a, n, q), (b_s, b_a, n, q)) \mid n < 7 \,\})\}
\end{aligned}
$$

- We now perform the second iteration of the algorithm. First the non-blocking predicate N is computed:

$$
\begin{aligned}
N((b_s, b_a, n, q_0)) &= \mathsf{true} \\
N((b_s, b_a, n, q_2)) &= \mathsf{true} \iff n < 7 \\
N((b_s, b_a, n, q_1)) &= \mathsf{true} \iff n < 6
\end{aligned}
$$

Then the bad predicate B is computed

$$
\begin{aligned}
B((b_s, b_a, n, q_0)) &= \mathsf{false} \\
B((b_s, b_a, n, q_1)) &= \mathsf{true} \iff n \geq 6 \\
B((b_s, b_a, n, q_2)) &= \mathsf{true} \iff n = 7
\end{aligned}
$$

Since the predicate B does is equivalent to the bad predicate computed in the previous iteration, C_2 will be equivalent to C_1, which implies that the algorithm terminates.

- Using the updated relation set C_1 and U we will construct the supervisor as seen in Figure 6.8.4.

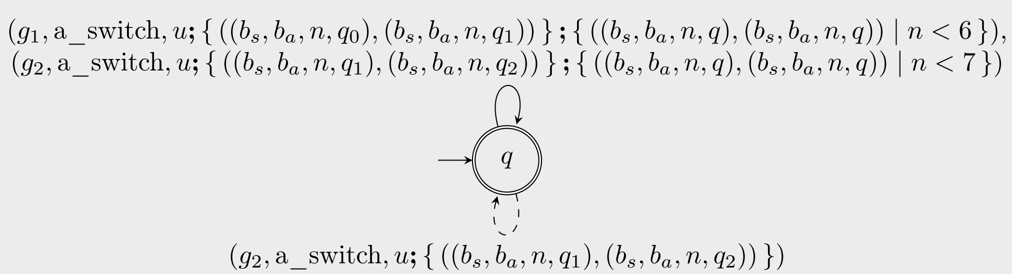

Figure 6.8.4: Most permissive supervisor for S.

Example 6.8.2 Suppose we have (refined) plant $\{0,1,2,3\}$-system P as shown in Figure 6.8.5 where

$$\begin{aligned}\text{player1Take} &= \{\,(c, c-1) \mid c \in \{\,0,\dots,5\,\}\,\}\\ &\cup \{\,(c, c-2) \mid c \in \{\,0,\dots,5\,\}\,\}\\ \text{player2Take} &= \{\,(c, c-1) \mid c \in \{\,0,\dots,5\,\}\,\}\\ &\cup \{\,(c, c-2) \mid c \in \{\,0,\dots,5\,\}\,\}\end{aligned}$$

$$\begin{aligned}g_1 &= \{\,(c,c) \mid c \in \{\,0,\dots,5\,\}, c > 0\,\}\\ g_2 &= \{\,(c,c) \mid c \in \{\,0,\dots,5\,\}, c = 0\,\}\\ N_0 &= \{\,3\,\}\end{aligned}$$

P essentially models a game where two players take turns in either taking one or two objects, where the player taking the last object loses the game. Player 1, controlled by the system, takes the first turn. Player 2 is controlled by the environment. The system enters its terminal state when player 1 wins the game. The game starts with 3 objects.

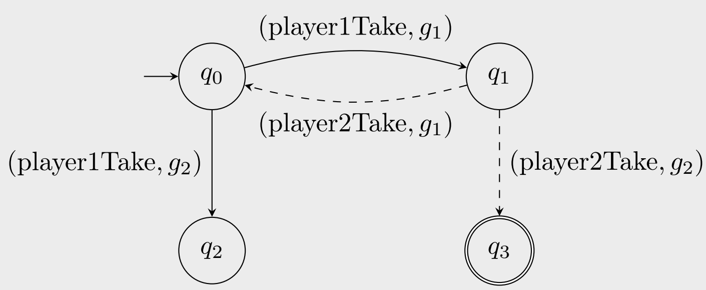

Figure 6.8.5: Visual representation of P.

We now compute the most-permissive supervisor for P using the complete algorithm.

- We first construct the sets C and U of binary relations on $D = (\{\,0,1,2,3\,\} \times \{\,q_0,q_1,q_2,q_3\,\})$.

$$\begin{aligned}U = \{&(id_D, \text{player2Take}, g_1; \{\,((n,q_1),(n,q_0))\,\}),\\ &(id_D, \text{player2Take}, g_2; \{\,((n,q_1),(n,q_3))\,\})\}\\ C = \{&(id_D, \text{player1Take}, g_1; \{\,((n,q_0),(n,q_1))\,\}),\\ &(id_D, \text{player1Take}, g_2; \{\,((n,q_0),(n,q_2))\,\})\}\end{aligned}$$

  We also construct the finalization predicate $F((n,q)) = q \in \{\,q_3\,\}$.

- We perform the first iteration of the algorithm. First the non-blocking predicate N is computed:

$$\begin{aligned}N((n,q_3)) &= \mathsf{true}\\ N((n,q_1)) &= \mathsf{true} \iff n < 3\\ N((n,q_0)) &= \mathsf{true} \iff n = 3\end{aligned}$$

Then the bad predicate B is computed

$$\begin{aligned}
B(n, q_2) &= \mathsf{true} \\
B(1, q_0) &= \mathsf{true} \\
B(3, q_1) &= \mathsf{true} \\
B(2, q_1) &= \mathsf{true}
\end{aligned}$$

We then update the controllable update relations:

$$\begin{aligned}
C_1 = \{&(id_D, \text{player1Take}, g_1; \{\, ((n, q_0), (n, q_1)) \,\}; \{\, ((n, q), (n, q)) \mid n \neq 2 \,\}), \\
&(id_D, \text{player1Take}, g_2; \{\, ((n, q_0), (n, q_2)) \,\}; \emptyset)\}
\end{aligned}$$

- We now perform the second iteration of the algorithm. First the non-blocking predicate N is computed:

$$\begin{aligned}
N((n, q_3)) &= \mathsf{true} \\
N((n, q_1)) &= \mathsf{true} \iff n < 3 \\
N((n, q_0)) &= \mathsf{true} \iff n = 3
\end{aligned}$$

  Then the bad predicate B is computed

$$\begin{aligned}
B(n, q_2) &= \mathsf{true} \\
B(1, q_0) &= \mathsf{true} \\
B(3, q_1) &= \mathsf{true} \\
B(2, q_1) &= \mathsf{true}
\end{aligned}$$

  Since the predicate B does is equivalent to the bad predicate computed in the previous iteration, C_2 will be equivalent to C_1, which implies that the algorithm terminates.

- Using the updated relation set C_1 and U we will construct the supervisor as seen in Figure 6.8.6.

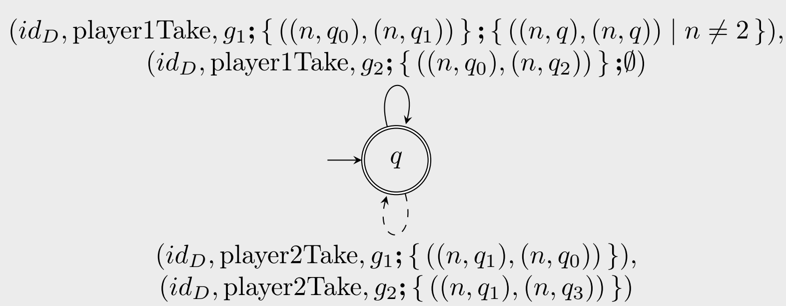

Figure 6.8.6: Most permissive supervisor for P

Part III

Language and Tooling

Chapter 7

Current Language and Toolchain

In this chapter we discuss the Compositional Interchange Format version 3 (CIF3), which consists of a language and a toolchain. The CIF3 language is a modeling language based on the EFA formalism as discussed in Section 5.3. In the CIF3 language, one can model a discrete event system (consisting of the plant and the requirements) in the form of multiple EFAs. Syntax is provided for defining the domain (using variables), states, events, guards, functions and transitions of an EFA. In a specification, multiple EFAs can be defined. In the toolchain, these EFAs are composed using the synchronous product operator. We discuss the language more in depth in Section 7.1. The toolchain contains tools for the simulation of the defined system using a given graphical representation of the modeled system, supervisory control synthesis and some code generation tools. We discuss the tools more in depth in Section 7.2. A more in depth description of the CIF3 language and toolchain can be found in [1].

7.1 Language Description

As previously discussed, the CIF3 language is based on the EFA formalism. Suppose we have EFA

$$A = (L, D, \Sigma, E, L_0, D_0, L_m)$$

where $D = D_1 \times \cdots \times D_k$. Each D_i is some type (e.g. integer, Boolean). We give the variable name d_i to the i'th tuple element of each element from D. For the set D_0 we have $D_0 = \{ (d_{1,0}, \ldots, d_{k,0}) \}$. The event set Σ is partitioned into the set of controllable events $\Sigma_c = \{ \sigma_c^1, \ldots, \sigma_c^n \}$ and the set of uncontrollable events $\Sigma_u = \{ \sigma_u^1, \ldots, \sigma_u^m \}$. This EFA can then be modeled in CIF3 in the following way.

```
automaton automatonName:
    controllable event $\sigma_c^1$
    ...
    controllable event $\sigma_c^n$

    uncontrollable event $\sigma_u^1$
    ...
    uncontrollable event $\sigma_u^m$

    disc $D_1$ $d_1$ = $d_{1,0}$
    ...
    disc $D_k$ $d_k$ = $d_{k,0}$

    // for each $l \in L$
```

```
    location $l$:
        initial; // when $l \in L_0$
        marked;  // when $l \in L_m$

        // for each $e \in E$ with $o_e = l$
        edge $\sigma_e$ when $g_e$ do $f_e$ goto $t_e$;

end
```

We can model EFA A_2 as given in Example 5.3.1 as follows.

```
automaton A2:
    controllable event a
    controllable event b

    disc int x = 0

    location loc0:
        initial;

        edge a when x < 3 do x := x + 1 goto loc0;
        edge a when x >= 3 goto loc1;
        edge b goto loc0;

    location loc1:
        marked;

        edge a goto loc1;
        edge b goto loc1;
end
```

The plant $P = P_1, \ldots, P_n$ and requirement $R_1, \ldots, R_m$ can be defined as follows.

```
plant $P_1$:
   // automaton definition
   ...
end
...
plant $P_n$:
   ...
end

requirement $R_1$:
   // automaton definition
   ...
end
...
requirement $R_m$:
   ...
end
```

Events and variables of the plant automata can be referred to in the requirement automata using the point notation. P and R are constructed using the synchronous product operator as discussed in Definition 5.3.2 (meaning $P = P_1 \,||\, \cdots \,||\, P_n$ and $R = R_1 \,||\, \cdots \,||\, R_m$). From P and R the refined plant with forbidden locations L_m can then be constructed according to Definition 5.3.3, which will be the input for Algorithm 2.

The CIF3 language is used in a number of projects. Two of which are the modeling of Lock III (a waterway lock consisting of a single chamber) [20], and the modeling of the Princess Marijke complex (which is a complex consisting of two waterway locks and a storm surge barrier) [21]. In these CIF3 models, the following issues regarding the CIF3 language can be observed.

- Defining a number of simple actuators (like buttons) is cumbersome, it would be more convenient if a list (or dictionary) of (indexed) buttons could be defined.
- A method to encapsulate subsystems might be useful. In the current situation, when accessing the state/event of some subsystem (for example a gate), then the user must check the states of the elementary actuators and sensors of the subsystem.
- The first issue described in Subsection 5.3.4 surfaces in the definition of user command automata. In the model an event is created for each user command.
- Single-state automata have to be created to model physical relations between separate plant components. This is also discussed in Subsection 5.3.4.

7.2 Toolchain Description

The CIF3 toolchain is developed in Java within the Eclipse Modeling Framework (EMF). The tools from EMF are used to model the abstract syntax of the CIF3 language. Based on this abstract syntax model, Java classes are generated which are used in the entire toolchain. This toolset consists of the following.

- A parser for the language described in Section 7.1.
- A type checker for functions and guards in CIF3 models.
- A simulator for CIF3 models. The user can provide a visual representation of the system in the form of an SVG file. The simulator can then animate this SVG file according to the variable values of the system.
- Implementation of the supervisory control algorithms for FSA (Algorithm 1), and EFA (Algorithm 2).
- Validation tools which test for the blocking, non-determinism, and controllability conditions.
- Tools for generating code from CIF3 Models. The following target languages are supported.
  - yED, which is a tool for graph drawing.
  - The verification languages mCRL2 and UPPAAL.
  - The general purpose languages Java and C.
  - Simulink, which is a MATLAB-based graphical modeling environment.
  - Programmable Logic Controller (PLC) code.

A potential problem of the current CIF3 toolchain, could be that the semantics of the CIF3 language is not made clear. An 'interpretation' of the language is implemented for each component. This lack of a central definition of the semantics, could lead to consistency, maintainability and extendibility problems. For example, suppose that some language extension must be implemented. After extending the metamodel, all interpretations of the metamodel (syntax) in the different components (for example, in the simulator and the supervisory control synthesis algorithm implementation) need to be adapted accordingly, which can lead to inconsistency.

Chapter 8

New Language and Tooling

In this chapter we will discuss the implementation of a proof of concept for the language and toolchain for X-Control. X-Control will be our DSL for modeling discrete event systems based on D-systems. We first discuss the approach we are going to apply when designing our language. We also discuss the toolchain, which consists of a simulator X-Control models, and an implementation of the supervisory control synthesis algorithm. Lastly, we will judge how extendible our language is by proposing a number of extensions.

8.1 Approach

In this section we discuss the approach which we are going to apply for designing X-Control. We will first discuss some background theory and possible approaches as discussed in [11].

8.1.1 Background

The two major aspects of designing formal languages, are *syntax* and *semantics*. The syntax of a DSL is usually defined in the form of a context-free grammar. There are multiple methods for defining semantics of DSLs. The method that we are going to discuss, is the *denotational* method. Denotational semantics consists of the following.

- The *semantic domain*, which is a collection of semantic values and operations. In CIF, the definition of the simulator could be seen as the description of the semantic domain.
- the *valuation function*, which is a mapping from the syntax to the semantics. This function essentially gives meaning to the syntax. In CIF, the interpretation of the metamodel for the simulator could be seen as a valuation function.

In order to put a DSL into practice, the syntax and semantics are expressed using a programming language. We then say this programming language is used as a *metalanguage*. The values of the semantic domain can be defined as value of types defined in the metalanguage. The operations of the semantic domain can be implemented as functions on these types. There are two implementation styles for DSLs. One being the *external DSL* style: a standalone language which is parsed and interpreted by the metalanguage. The other being the *internal DSL* style (also called the *embedded* style), which exists in the metalanguage itself. For describing syntax using the internal DSL style in the metalanguage, there are two options:

- *Deep embedding*, where the syntax is explicitly represented by a data type. The constructors represent the grammar productions of the language.

- *Shallow embedding*, where the constructors of the semantic domain are used for operations of the DSL. Function definitions are introduced for operations that are not directly represented by the constructors of the semantic domain. Syntax described in this style is relatively easy to modify, which makes it especially useful when the language is still frequently changing.

For designing languages the authors of [11] discuss the following two approaches.

- The syntax-driven approach: first the syntax of the languages is designed, then the semantic domain is constructed. This is the more traditional approach.

- The semantics-driven approach: first the semantic domain of the languages is constructed, then syntax is designed for this semantic domain.

In the following sections we discuss both approaches more in depth.

8.1.2 Syntax-driven design

When applying the syntax-driven design approach, one starts with enumerating the features the language should have. For example, in the case of a calendar DSL, features as adding, moving and deleting appointments are denoted. Syntax is then designed for these features. After the syntax is designed, its semantic domain is defined. This means the types of the domain values and the operations on said values are defined. Lastly, a valuation function is constructed for mapping abstract syntax values to semantic domain values.

This approach is clearly a more feature-driven approach. An advantage of this approach is that we end up with a semantic domain that works very well for the syntax. This implies that it also implements the features enumerated in the beginning of the process. According to the authors of [11], a major disadvantage of this approach is that the resulting language design will be rigid, meaning that future extensions will be difficult to implement.

8.1.3 Semantics-driven design

When applying the semantics-driven design approach, one starts with identifying and implementing a small and compositional semantics core. This approach forces language designers to carefully consider the essence of what their language represents at the start of designing process.

The semantic driven design process for some domain D consists of the following steps.

1. Decompose the domain D into subdomains $D_1, D_2, \ldots$, and establishing the relationships between these domains.

2. Model the decomposed semantic domain in the metalanguage. Each subdomain forms the basis for a *micro DSL*. The identified relationships between subdomains are modeled as *language schemas*. An example of such a language schema would be a mapping for establishing a relationship between instances of types from two different micro DSLs.

3. Design the syntax. This step can also be broken down into two steps:
   (a) Construct the syntax for the micro DSLs.
   (b) Construct the *domain integration syntax*. Domain integration syntax represents higher-level operations of our DSL. Such operations cover multiple micro DSLs.

The authors of [11] advocate for this approach, since it leads to a more compositional language design (if applied correctly), which are more general and reusable, and less ad hoc.

8.1.4 Our Approach

Based on the findings in [11], we will follow the following approach for designing X-Control.

- We apply the syntax driven approach, since according to the authors it leads to a more compositional design.
- We will implement our language using the internal DSL style, since implementing and modifying internal DSLs is relatively easy.
- For our metalanguage we will use Haskell, since it well established and often used as a metalanguage.

8.2 Semantic Domain of X-Control

In this section we discuss the semantic domain of X-Control. We discuss all subdomains of our DSL, which we implement as types in our metalanguage. These subdomains correspond to the mathematical constructs discussed in Chapter 6. The implementations of these constructs should correspond with their mathematical notations. We establish relations between the subdomains using type variables in Haskell.

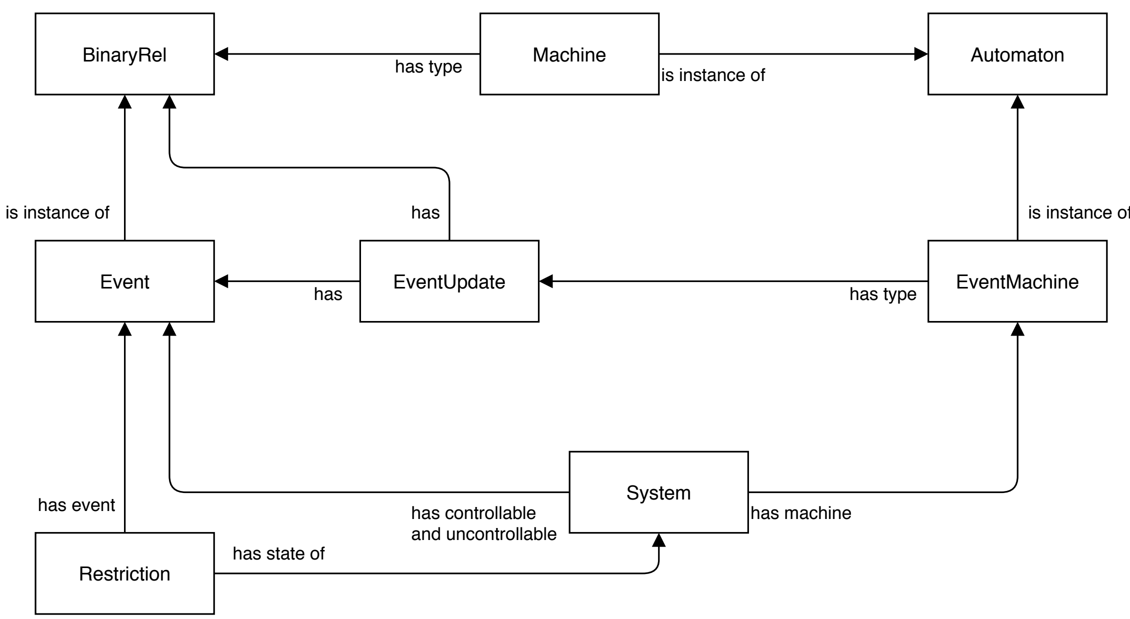

Figure 8.2.1: Diagram showing relations between the subdomains

An overview of the relations between our subdomains is shown in Figure 8.2.1. In the follow subsections we discuss the separate subdomains, in order of occurrence in Chapter 6. A more detailed description of the implementation of the semantic domain can be found in Appendix C.

8.2.1 Automaton

The subdomain *Automaton* considers the definitions discussed in Chapter 3. The Σ-automaton (Q, I, T, δ) as introduced in Definition 3.1.1 is modeled as

```haskell
data Automaton a b
  where
    Automaton
        :: ( AutomatonType a
           , Eq b
           )
        => [b]               -- states
        -> [b]               -- initial states
        -> [b]               -- terminal states
        -> [a]               -- alphabet
        -> (b -> a -> [b]) -- transition relation
        -> Automaton a b
```

Instead of a tuple, we implement the automaton construct as a data type. The type variable `a` represents the type of the elements of the alphabet Σ. The first three parameters represent the sets Q, I, and T. The fourth parameter represents the alphabet Σ, which is added since we cannot easily define a Haskell class that enforces a finite type domain. We use list constructs for these sets, since it is easier to work with lists than to work with sets in Haskell. A transition relation is then modeled as a function of type `b -> a -> [b]`. This allows for defining the transition relation in a similar style as shown in part II. The automaton from Example 3.1.3 can then be defined as follows.

```haskell
automatonEx3 :: Automaton Char Int
automatonEx3 = Automaton qs is ts alph (==>)
  where
    qs = [0, 1, 2]
    is = [0]
    ts = [0, 1]
    alph = ['a', 'b']
    0 ==> 'a' = [1]
    0 ==> 'b' = [2]
    1 ==> 'a' = [2]
    1 ==> 'b' = [0]
    2 ==> 'a' = [2]
    2 ==> 'b' = [2]
```

We also implement *free monoids* from Definition 3.2.1, using a simple *snoc-list* structure (where the last element of the list is accessible, instead of the first as in the *cons-list* structure).

```haskell
data FreeMonoid a = Empty | FreeMonoid a :> a

instance Semigroup (FreeMonoid a) where
fm <> Empty = fm
fm1 <> (fm2 :> x) = (fm1 <> fm2) :> x

instance Monoid (FreeMonoid a) where
mempty = Empty
```

We then define the following operations on automaton.

- `gamma :: AutomatonType a => automaton a b -> [b] -> FreeMonoid a -> [b]`, which is an implementation of the operation γ from Definition 3.2.4.
- `checkAccept :: (AutomatonType a, Eq b) => Automaton a b -> FreeMonoid a -> Bool`, which checks if some $\omega \in \Sigma$ is in $\mathcal{L}(A)$ using the equation $\mathcal{L}(A) = \{\, \omega \mid \omega \in \Sigma^*, I\omega \cap T \neq \emptyset \,\}$.
- `getBehavior :: Automaton a b -> [(b, FreeMonoid a)]`, which returns all $\omega \in \mathcal{L}(A)$. Each ω is combined with the corresponding terminal state identifier.

8.2.2 Relations and Events

Binary relations are used in EventMachines, and they are used to model the events for a *D*-System. These binary relations are represented by the subdomain *BinaryRel*, which is implemented as follows.

```
data BinaryRel a = BinaryRel String (a -> [a])

identityRel :: BinaryRel a
identityRel = BinaryRel "id" (:[])

instance AutomatonType (BinaryRel a)

instance Eq (BinaryRel a)
  where
    (BinaryRel label1 rel1) == (BinaryRel label2 rel2) = label1 == label2

instance Semigroup (BinaryRel a)
  where
    (BinaryRel label1 rel1) <> (BinaryRel label2 rel2)
      =
        BinaryRel (label1 ++ ";" ++ label2) (rel1 >=> rel2)

instance Monoid (BinaryRel a)
  where
    mempty = identityRel
```

Binary relations are given a label for identification. Just as with the transition relation, the actual relation is defined as a function `a -> [a]`. We then make `BinaryRel` an instance of `Monoid`, which can be easily done using the operator >=> from the list monad instance.

The subdomain *Event*, corresponding with Definition 6.2.2 is simply an instance of *BinaryRel*.

```
type Event a = BinaryRel a
```

8.2.3 EventMachines

We first define a separate domain for the labels (ϕ, e, ϕ') on the transitions of an EventMachine.

```
type = EventUpdate a = (BinaryRel a, Event a, BinaryRel a)
```

Wherafter we define the subdomain *EventMachine* corresponding with Definition 6.2.2.

```
type EventMachine a = Automaton (EventUpdate a) StateLabel
```

The type of the elements of the alphabet is in this case `EventUpdate a`. The type of the state identifier is `StateLabel`. This datatype is introduced to ease the implementation of the synchronous product operator later on.

8.2.4 *D*-Systems and Restrictions

We now have all our ingredients to define the subdomain *System* with corresponds with the definition of *D*-systems as given in Definition 6.2.5.

```
data System a = System
    { machine :: EventMachine a
    , controllableEvents :: [BinaryRel a]
```

```
    , uncontrollableEvents :: [BinaryRel a]
    , domain :: [a]
    , initialValues :: [a]
    }
```

We use field labels for easier access to the parameters' values. We once again use lists instead of sets for the same reason as for the *Automaton* subdomain. We also add a list of domain elements as a parameter, since we cannot easily enforce a finite type domain (as with the automaton alphabet).

The sensor from Example 6.5.1 can then be modeled as follows.

```
sensorSwitchEvent = BinaryRel "sensorSwitch" rel
  where
    rel b = [not b]

sensorSystem :: System Bool
sensorSystem = System sensorMachine [] [sensorSwitchEvent] [False, True] [False]
  where
    sensorMachine = Automaton qs is ts phis delta
      where
        offState = SingleLabel "sensorOff"
        onState = SingleLabel "sensorOn"

        qs = [offState, onState]
        is = [offState]
        ts = [offState]
        phis = [(identityRel, sensorSwitchEvent, identityRel)]
        delta q t = getStateLabel q ==> show t

        "sensorOff" ==> "(id,sensorSwitch,id)" = [onState]
        "sensorOn"  ==> "(id,sensorSwitch,id)" = [offState]
```

We then define the following operations on the subdomain *System*.

- `getTraces :: System a -> [(FreeMonoid (BinaryRel a), a)]`, which returns a list of possible event traces in our system, together with the corresponding domain values.
- `syncEventSystems :: DomainComposition d1 d2 dc -> System d1 -> System d2 -> System dc` , which implements the synchronous product operator as introduced in Definition 6.3.4. For this operation a `DomainComposition d1 d2 dc` must be given. An instance of this type implements a domain composition ($\otimes$), where `d1` corresponds with domain D_1, `d2` with D_2, and `dc` with the composed domain $D_1 \otimes D_2$. `DomainComposition` is defined as follows.

```
data DomainComposition d1 d2 dc = DomainComposition
    { combine :: d1 -> d2 -> dc
    , decompose :: dc -> (d1, d2)
    , checkComp :: d1 -> d2 -> Bool
    , extract1 :: dc -> d1
    , extract2 :: dc -> d2
    , augment1 :: dc -> d1 -> dc
    , augment2 :: dc -> d2 -> dc
    }
```

  - `combine`: A mapping from instances of the two original domains to an instance of the composed domain (according to the composition).
  - `decompose`: A mapping from an instance of the composed domain to the instances of the original domains.

- `checkComp`: Check if the instances of the two original domains can be mapped to the composed domain (in most cases, this would be checking of the shared domain values are equal).
- `extract1`: A mapping from an instance of the composed domain to the instance of the first original domain.
- `extract2`: A mapping from an instance of the composed domain to the instance of the second original domain.
- `augment1`: Suppose we have a value `vc` of the composed domain corresponding to the values `v1` of the first domain and `v2` of the second domain, and a value `v1'` of the first domain. `augment1` maps `vc` and `v1'` to the instance of the composed domain corresponding with `v1'` and `v2`.
- `augment2`: Suppose we have a value `vc` of the composed domain corresponding to the values `v1` of the first domain and `v2` of the second domain, and a value `v2'` of the second domain. `augment2` maps `vc` and `v2'` to the instance of the composed domain corresponding with `v1` and `v2'`.

- `(\/) :: System d -> [Restriction d] -> System d`, which implements the restrict operator (↓) as introduced in Definition 6.6.2. The second argument of this operator is a list of element of the subdomain *restriction*, which corresponds with Definition 6.6.1 and is defined as follows.

```haskell
type Restriction a = (Event a, StateLabel)
```

- `synthesizeSupervisor :: (Eq d, Show d) => System d -> System (d, StateLabel)`, which implements the supervisory control synthesis algorithm as defined in Section 6.8. Given a refined plant P, the function will return a maximally permissive supervisor S.
- `supervise :: (Eq d, Show d) => System d -> System (d, StateLabel)`, which, given a refined plant P, computes the supervisor S using `synthesizeSupervisor`, and returns the plant synchronized with the supervisor $P \,||\, S$.

Values of the the subdomain *System* are the most important expressible values. Models for a discrete event system consists of multiple *System* values which will be composed into one single *System* value using synchronization (as discussed in Section 6.5). This value can then be applied to the supervisory control synthesis algorithm, or simulated using the simulator (which we will discuss in Section 8.4).

8.3 Syntax of X-Control

In semantics-driven design, we systematically construct syntax for the subdomains in the semantic domain. In our case we define syntax for the types defined in Section 8.2. As discussed in Section 8.1, we will implement the syntax as a mostly shallow embedded *internal DSL* in Haskell. In a pure shallow embedding, only the data types and their constructors defined in the semantic domain are used in the syntax. This implies that, should we implement a pure shallow embedding, then we may not introduce new datatypes for our syntax. Because this method of defining syntax is rather restrictive we will slightly deviate from the shallow embedding method, as we introduce some auxiliary data types to construct our internal syntax. The constructors of these data types act as keywords of our syntax, along with a set of (string) constants. We describe syntax for the following elements of our semantic domain.

- Automata, where the syntax can be used to define automaton with arbitrary label type `a`. Since EventMachines are automata with label type $\mathcal{P}(D^2) \times (E_c \cup E_u) \times \mathcal{P}(D^2)$ (according to Definition 6.2.2), we can also use this syntax to define EventMachines.
- Domains (as in the domain D of some D-systems).
- Binary relations. This syntax will then be used to describe the *events* in a system, since events are modeled as binary relations in our semantic domain.
- D-systems. This syntax allows the end user to specify the components of a D-system (the domain, the events, the machine, etc.).
- Synchronization and restrictions, of which the syntax is given in the form of *Modules*. Modules can be interpreted as a synchronous product of a set of systems, followed by a restriction.

In the following subsections we will discuss the syntax for these elements in more detail. A more detailed description of the implementation of the syntax can be found in Appendix D.

8.3.1 Automata

Syntax for defining automaton, of which the alphabet is of a given type `a`, consists of a (Haskell) list of the following possible statements.

- `State "stateName"`, which declares a regular non-initial and non-terminal state.
- `InitialState "stateName"`, which declares an initial state (which is not terminal).
- `TerminalState "stateName"`, which declares a terminal state (which is not initial).
- `InitialTerminalState "stateName"`, which declares a state which is both initial and terminal.
- `Edge from "originStateName" to "targetStateName" with` *`symbol`*, which declares a transition edge from the state named `"originStateName"` to the state named `"targetStateName"` with a symbol. *`symbol`* is an instance of type `a` and is part of the alphabet of the automaton.

Note that, semantically, the order in which the declarations occur in the list does not matter. The keywords starting with a capital letter (`State`, `InitialState`, etc.) are all constructors of the datatype `AutomatonDeclaration`. The other keywords (`from`, `to`, `with`) are string constants which are given as (dummy) parameter values to the constructor.

The valuation function for automaton `automaton :: [AutomatonDeclaration a] -> Automaton a StateLabel`, transforms the list of statements to an automaton instance of our semantic domain. The automaton from Example 3.1.3 can then be defined as follows,

```
exampleAutomaton = automaton [
    InitialState "q0",
    State "q1",
    TerminalState "q2",

    Edge from "q0" to "q1" with 'a',
    Edge from "q0" to "q2" with 'b',
    Edge from "q1" to "q0" with 'b',
    Edge from "q1" to "q2" with 'a',
    Edge from "q2" to "q2" with 'a',
    Edge from "q2" to "q2" with 'b'
    ]
```

8.3.2 Domains

In most cases, a system's domain D consists of a number of subdomains $D_1, \ldots, D_n$ where $D = D_1 \times \cdots \times D_n$. Each subdomain can be considered as a *domain element* or *domain variable*. In practice it is useful to give a name to each domain element. We call such a name an *element identifier*. For now we will introduce syntax for declaring Boolean and integer domain elements with an element identifier. This syntax is defined as follows.

- `BoolElement "elementId"` *`intialValue`*, which is a declaration of a Boolean element with the given element identifier and an initial (Boolean) value.
- `IntElement "elementId"` *`range initialValue`*, which is a declaration of an integer element with the given element identifier, a range of possible (integer) values which this element can have, and a initial (integer) value. We define a range of possible values for each integer element, since for the supervisory control synthesis algorithm, the given D-system must have a finite domain.

The valuation function `declareDomain` transforms a list of element declarations to a mapping from the element identifiers to the (initial) values. The function `getPossibleDomainValues` transforms a list of element declarations to a list of possible mappings from the element identifiers to the element values (based on the given ranges). In the context of D-systems, this list forms the domain D of the system. An example domain could be defined as follows:

```
exampleDomain = [BoolElement "exampleBool" False,
                  IntElement "exampleInt" [0..5] 5
                 ]
```

8.3.3 Binary Relations

Binary relations are used to model the events and other operations of a D-system. In a binary relation on D, an instance of D may be related to zero or more other instances of D. We will provide syntax in X-Control to define relations in a 'procedural way', which is more convenient for end users which are not familiar with functional languages. This can be achieved when using the '`State a`' monad. The state monad allows us to describe a sequence of manipulations on some instance of the type `a` using so called *do-notation* in Haskell. In our case the type '`a`' would be the mapping from element identifiers to values. By wrapping the state monad in a newly defined type (which we call the `DomainState` monad), we can hide this (relatively) complicated type from the end user. The following functions can be used in this do-notation, and will be part of our syntax.

- `getBoolValue "elementId"`, which returns the current (Boolean) value of the Boolean element with the given element identifier.
- `getIntValue "elementId"`, which returns the current (integer) value of the integer element with the given element identifier.
- `setBoolValue "elementId"` *`boolValue`*, which sets the Boolean element with the given element identifier with the given (Boolean) value.
- `setIntValue "elementId"` *`intValue`*, which sets the integer element with the given element identifier with the given (integer) value.

Do-notation can then be used in our syntax in the follow way.

```
do { ...
     b <- getBoolValue "exampleBool";
     x <- getIntValue "exampleInt";
     ...
     setBoolValue "exampleBool" (not b);
     setIntValue "exampleInt" (x + 5);
     ...
   }
```

We have the following syntax for different types of relation declarations.

- `Function "functionName" $ do {...}`, which is a relation where each instance of the domain is related to exactly one instance (which may be the same instance). An example could be defined as follows.

```
Function "exampleFunction" $ do {
    b <- getBoolValue "exampleBool";
    x <- getIntValue "exampleInt";

    if b then
        setIntValue "exampleInt" (x - 1);
    else
        setIntValue "exampleInt" (x + 1);
    }
```

- `Guard "exampleGuard" $ do {...; return` *`boolExpression`*`;}`, which is a relation where each instance of the domain is either related to the same instance or to no instance at all. The instance relates to itself in the *guard* if and only if *`boolExpression`* evaluates to `True` for this instance. An example could be defined as follows.

```
Guard "exampleGuard" $ do {
    b <- getBoolValue "exampleBool";
    x <- getIntValue "exampleInt";

    return (b || (x > 2));
}
```

- `Relation "relationName" $ do {...; return [do {...}, do {...}, ...];}`, which is a relation where each instance of the domain can be related to 0 or more instances. Each *do-block* in the returned list describes how these instances are established. An example could be defined as follows.

```
Relation "exampleRelation" $ do {
    x <- getIntValue "exampleInt";

    return [
        do { setBoolValue "exampleBool" False; setIntValue "exampleInt" (x + 1); },
        do { setBoolValue "exampleBool" True; setIntValue "exampleInt" (x - 1); }
    ];
}
```

  Note that the syntax for *relations* can also be used to define *functions* and *guards*.

The valuation function `declareRel` transforms an instance of one of the aforementioned relation declarations to an instance of the binary relation type as defined in the semantic domain.

8.3.4 Systems

We now introduce syntax for D-systems. In order to define a D-system $S = (M, E_c, E_u, D_0)$, one has to define the domain D, the D-EventMachine M, the controllable and uncontrollable events E_c and E_u, and the initial values D_0. To enable users to provide these components, we make use of the *field-labels* notation from Haskell in our syntax. A D-system can be defined as follows.

```
SystemSpecification
   { domainElements = [...],
     controllableEvents = [...],
     uncontrollableEvents = [...],
     otherOperations = [...],
     machine = [...]
   }
```

The fields of this `SystemSpecification` can be defined as follows.

- `domainElements`: The list of domain elements as described in Subsection 8.3.2. This field describes both the domain and the initial values of the system.
- `controllableEvents`: The list of controllable events of the system. Each event is defined as a binary relation using the syntax described in Subsection 8.3.3.
- `uncontrollableEvents`: The list of uncontrollable events of the system. Just as with controllable events, each event is defined as a binary relation.
- `otherOperations`: A list of relations not modeling events, which can be used in the D-EventMachine of the system (e.g. guards and update functions).
- `machine`: The definition of the D-EventMachine of the system, which is described using the automaton declaration syntax described in Subsection 8.3.1. The symbols of the automaton's alphabet are of the form `("otherOperationName1", "eventName", "otherOperationName2")`, where `otherOperationName1` and `otherOperationName2` are names of relations defined in the `otherOperations` list, and `eventName` is the name of a relation defined in either the `controllableEvents` list or the `uncontrollableEvents` list. The identity relation (relation in which every instance of the domain relates to itself) is always accessible via the relation name `"id"`.

The valuation function `declareSystem` transforms a given `SystemSpecification` to a D-system. An example modeling a simple actuator could be defined as follows.

```
actuator = declareSystem SystemSpecification
   { domainElements = [
         BoolElement "actuatorStatus" False
     ],
     controllableEvents = [
         Function "switchActuator" $ do {
             actuatorStatus <- getBoolValue "actuatorStatus"
             setBoolValue "actuatorStatus" (not actuatorStatus);
             }
     ],
     uncontrollableEvents = [],
     otherOperations = [],
     machine = [
         InitialTerminalState "actuatorOff",
         State "actuatorOn",

         Edge from "actuatorOff" to "actuatorOn" with
```

```
            ("id", "switchActuator", "id"),
        Edge from "actuatorOn" to "actuatorOff" with
            ("id", "switchActuator", "id")
    ]
  }
```

The plant P in Example 6.8.2 can be defined as follows.

```
systemExample = declareSystem SystemSpecification
    { domainElements = [
          IntElement "coins" [0..5] 5
      ],

      controllableEvents = [
          Relation "player1Take" $ do {
          coins <- getIntValue "coins";

          return [
              setIntValue "coins" (coins - 1),
              setIntValue "coins" (coins - 2)
          ];
          }
      ],

      uncontrollableEvents = [
          Relation "player2Take" $ do {
              coins <- getIntValue "coins";

              return [
                  setIntValue "coins" (coins - 1),
                  setIntValue "coins" (coins - 2)
              ];
              }
      ],

      otherOperations = [
          Guard "notGameOver" $ do {
              coins <- getIntValue "coins";
              return $ coins > 0;
              },
          Guard "gameOver" $ do {
              coins <- getIntValue "coins";
              return $ coins == 0;
              }
      ],

      machine = [
          InitialState "player1Turn",
          State "player2Turn",
          State "player1Lost",
          TerminalState "player2Lost",

          Edge from "player1Turn" to "player2Turn" with
              ("id", "player1Take", "notGameOver"),
          Edge from "player1Turn" to "player1Lost" with
              ("id", "player1Take", "gameOver"),
          Edge from "player2Turn" to "player1Turn" with
              ("id", "player2Take", "notGameOver"),
          Edge from "player2Turn" to "player2Lost" with
```

```
                ("id", "player2Take", "gameOver")
        ]
    }
```

8.3.5 Modules

The synchronous product operator which is discussed in Section 6.3 and implemented in the semantic domain allows us to break up a discrete event system into multiple components, and create a separate D-system for each component. Moreover, the restriction operator as introduced in Section 6.6 (and which is also implemented in the semantic domain) allows one to remove behavior from a D-system which is not expected to happen in physical instances of the system, due to physical relations between subcomponents.

The concepts of *synchronization* and *restrictions* are both comprised by the *module* syntax. A module is a list of the following possible declarations.

- `DeclareSystem "systemName" systemSpecification`, which declares a system with a name to identify the system. `systemSpecification` is a system specification using the syntax described in Subsection 8.3.4. Domain elements and events from other systems can be accessed via point notation (`"sysName.elementName"` for elements and `"sysName.eventName" for events`).
- `DeclareRestriction "sysName1.stateName" restricts "sysName2.eventName"`, which declares a restriction. This means that, in this instance, if the system with name `"sysName1"` is in state with name `"stateName"`, then the event with name `"eventName"` belonging to system with name `"sysName2"` cannot occur.

The valuation function `declareModule` transforms a given module, and transforms it to a single D-system. This function makes use of the valuation function for system specifications, the synchronous product operator, and the restrict operator.

The start of an incomplete example module containing a sensor and an actuator, which we will complete in two different ways, can be defined as follows.

```
exampleModule = declareModule [
    DeclareSystem "actuator" SystemSpecification
        { domainElements = [
                BoolElement "actuatorStatus" False
            ],
            controllableEvents = [
                Function "switchActuator" $ do {
                    actuatorStatus <- getBoolValue "actuatorStatus";
                    setBoolValue "actuatorStatus" (not actuatorStatus);
                    }
            ],
            uncontrollableEvents = [],
            otherOperations = [],
            machine = [
                InitialTerminalState "actuatorOff",
                State "actuatorOn",

                Edge from "actuatorOff" to "actuatorOn" with
                    ("id", "switchActuator", "id"),
                Edge from "actuatorOn" to "actuatorOff" with
                    ("id", "switchActuator", "id"),
            ]
```

```
        },

        DeclareSystem "sensor" SystemSpecification
            { domainElements = [
                  BoolElement "sensorStatus" False
              ],
              controllableEvents = [
                  Function "switchSensor" $ do {
                      sensorStatus <- getBoolValue "sensorStatus";
                      setBoolValue "actuatorStatus" (not sensorStatus);
                  }
              ],
              uncontrollableEvents = [],
              otherOperations = [],
              machine = [
                  InitialTerminalState "sensorOff",
                  State "sensorOn",

                  Edge from "sensorOff" to "sensorOn" with
                      ("id", "switchSensor", "id"),
                  Edge from "sensorOn" to "sensorOff" with
                      ("id", "switchSensor", "id"),
              ]
            },
        ...
```

We can model the requirement as discussed in Example 6.5.1 by completing the model in the following way.

```
    ...
    DeclareSystem "requirement" SystemSpecification
        { domainElements = [],
          controllableEvents = [],
          uncontrollableEvents = [],
          otherOperations = [],
          machine = [
              InitialTerminalState "r0",
              State "r1",

              Edge from "r0" to "r1" with ("id", "sensor.switchSensor", "id"),
              Edge from "r1" to "r1" with ("id", "sensor.switchSensor", "id"),
              Edge from "r1" to "r0" with ("id", "actuator.switchActuator", "id")
          ]
        }
]
```

Alternatively, we can add a restriction which prevents the "switchSensor" event when the actuator is "actuatorOff".

```
    ...
    DeclareRequirement "actuator.actuatorOff" restricts "sensor.switchSensor"
]
```

8.4 Tooling for X-Control

In this section we describe the tooling for X-Control, which consists of the following.

- A method to describe discrete event system with the syntax of X-Control.
- The simulation of systems defined in X-Control.
- Applying supervisory control synthesis for systems defined in X-Control.

8.4.1 Describing Systems

The tooling for X-Control requires the Glasgow Haskell Compiler [3] (GHC). The description of a discrete event system using X-Control is to be done in a Haskell file (.hs). In this Haskell file, a number of modules have to be imported. This is to be done as follows.

```
1 import SemanticDomain
2 import Syntax
3 import Simulator
4
5 system = declareModule [...]
```

The description of the system using the syntax of X-Control starts at line 5. The module is then to be loaded in ghci (the interactive environment of GHC) using `ghci` *`filename`*`.hs`.

8.4.2 Simulation

The simulator can be used to demonstrate a system modeled in X-Control. The controllable events of the system are randomly chosen by the simulator (if enabled) at timer intervals of 1 second. The uncontrollable events are initiated by the user.

When the module is loaded into ghci, a D-system can be simulated with the command `simulateSystem` *`systemName`*. The following will be shown in the command prompt.

```
State: currentStateName
Current value: currentDomainValue
```

Where `currentStateName` is the name of the current state, and `currentDomainValue` is the textual representation of the current domain value. This value changes every transition according to the relations and event on the transition's edge.

The user can initiate an uncontrollable event by pressing '`e`' on the keyboard during the simulation. If there are currently no enabled uncontrollable event, then `"No enabled uncontrollable events"` is shown in the command prompt. Otherwise, the list of enabled uncontrollable events will be enumerated in the prompt. The user can then choose a uncontrollable event by entering its respective number. After this, a list of possible target states and domain value combinations is enumerated in the prompt. These combinations are reached via transition with the chosen event. Again, the user chooses a combination and the simulation proceeds. The user can stop the simulation by pressing '`q`'.

Further documentation of the simulator can be found in Appendix E.

8.4.3 Supervisory Control Synthesis

In the semantic domain, a function is defined which performs supervisory control synthesis for a given D-system. The resulting supervisor can then be synchronized with the original system to obtain the supervised system. The function `supervise` will execute both steps (supervisory control synthesis and synchronization) for a given D-system, and returns the result. This function is to be used as follows in ghci.

```
supervisedSystem = supervise system
```

8.5 Extendibility

In this section we propose a set of possible extensions for X-Control. For each proposal, we also briefly discuss a possible implementation. This should give us an idea on the extendibility of X-Control.

8.5.1 Parameterized Systems

From a reusability perspective, it would be useful to give a parameterized definition of a system. This definition can then be used in some module, given appropriate values for the parameters.

For this functionality we do not have to change the syntax, nor the semantics. One can define a parameterized system as a Haskell function in the following way.

```
exampleParameterizedSystem param1 ... paramN = SystemSpecification ...
```

These parameterized definitions can then be used in models as follows.

```
exampleModule = [
    ...
    DeclareSystem "systemName" (exampleParameterizedSystem value1 ... valueN),
    ...
    ]
```

8.5.2 Lists of Systems

Suppose we have some parameterized system. In some cases a large number of instances of this system are needed. For this reason, it would be useful to be able to define a list of instances of the parameterized system. Events and domain elements of these instances can be referred to by using the name of the list together with the index of the relevant instance. An example of a situation where lists of systems would be useful, is an array of n identical light sources (like LEDs).

To implement this extension, we could add the following declaration statement for modules.

```
DecelareSystemList "listName" systemSpecification n
```

From this declaration, n copies of `systemSpecification` should be created.

In the valuation function `declareModule`, each system specification in the list can then be given the name `listname[i]`, where i is the index the respective system. These system specifications can then be added to the list of other system specifications.

We can use the example parameterized system from the second subsection as an example.

```
DeclareSystemList "systems" (exampleParameterizedSystem value1 ... valueN) n
```

From this declaration, a copy of `systemSpecification`, where the specification has a parameter `p`, should be created for every value for the parameter in the given list.

```
DecelareParameterizedSystemList "listName" (\p -> systemSpecification) [p1, ..., pn]
```

For example, suppose we have the following parameterized system, where the event `countEvent` occurs n times.

```
counterSystems n = System Specification
    { domainElements = [
        IntElement "counter" [0..n] 0
      ],
      controllableEvents = [],
      uncontrollableEvents = [
        Function "countEvent" $ do {
            i <- getIntValue "counter";
            setIntValue "counter" (i + 1);
        }
      ],
      otherOperations = [
        Guard "lessThanTarget" $ do {
            i <- getIntValue "counter";
            return (i < n);
        }
        Guard "reachedTarget" $ do {
            i <- getIntValue "counter";
            return (i == n);
        }
      ],
      machine = [
        InitialState "q0",
        TerminalState "q1",

        Edge from "q0" to "q0" with ("id", "countEvent", "lessThanTarget"),
        Edge from "q0" to "q1" with ("id", "countEvent", "reachedTarget")
      ]
}
```

A list of 10 systems, where the system with index i has the event occurring i times, can then be declared as follows within a module.

```
DeclareParameterizedSystemList "counters" counterSystems [0..9]
```

In the valuation function `declareModule`, we can use `map` to compute the list of system specifications, where the specification with index i is the parameterized specification with the ith value of the given list applied. We can then use the aforementioned method of naming the system specifications in the list.

8.5.3 Nested Modules

For complex systems it may be useful to define the sub components in a hierarchical manner. This means that each subcomponent P_i can have its own sub-subcomponents $P_{i,1}, \ldots, P_{i,m}$. For our language this would translate to *nested modules*, which means that modules do not contain only system and restriction declaration, but (sub-)module definitions as well.

To implement this we need to introduce a new declaration statement for modules.

```
DeclareModule "moduleName" [declarationSystems]
```

In the valuation function `declareModule`, the names of the systems have to be renamed using the point notation (by adding the prefix `moduleName.` to the system names). One can then use the point notation to refer to events and variables from systems in the submodules using this point notation. By doing this renaming process in a recursive way, the modules hierarchy can then be flattened into a single list of systems and restrictions.

8.5.4 Event Aliases

Suppose we have the nested modules as described in Subsection 8.5.3. Suppose we have a module `module1`, which contains a submodule `module1A`. `module1A` contains a system `systemA1` with event `event`. If we want to refer to `event` in another system in `module1` using the point notation, then we have the reference `module1A.systemA1.event`. Since the notation of this reference is quite long it would be useful to define an alias for event `systemA1.event` in `module1A`. Suppose we define this alias as `eventX`, then this event can be referred to in `module1` as `module1A.eventX`.

For this extension, we add the following module declaration statement.

```
DeclareAlias "eventName" "eventAlias"
```

For the valuation function `declareModule`, we can then create a recursive function to create a mapping from alias to event name. This function can then be used to replace the aliases used in event references with the full event names.

8.5.5 Boolean Expression in EventMachine Labels

As discussed in Subsection 8.3.3, we can define binary relations function as transition guards using the `Guard` keyword. These guards could be reused if Boolean expressions with the guard names as identifiers can be used in the transition edges of EventMachines. It would also be useful if these identifiers could also point to Boolean domain elements.

To implement this one could simply write a simple parser for Boolean expressions. In [2] it is shown how this can easily be done with the Parsec package. Using the `State a` monad as described in Subsection 8.3.3, we transform these Boolean expressions to binary relations. These relations will then essentially evaluate these expressions.

Suppose we have the guards `guard1` and `guard2` in our system, we could then do the following in our machine.

```
Edge from "state1" to "state2" with ("!(guard1 || guard2)", "eventName", "id")
```

8.5.6 Requirements Based on Formulae

In many cases requirements can be formulated in a more compact way (as in more compact than a separate system) by using a formula. In these cases we say that a event may only occur if some Boolean expression (on the domain elements of the system(s)) is satisfied.

For this extension, we add the following module declaration statement.

```
DeclareRequirementFormula "eventName" requires $ do {
    ...;
    return booleanExpression;
    }
```

Such a requirement declaration could then be transformed to a system referring to the given event in the valuation function `declareModule`. This system's machine then has only one state, with a single self loop transition. This transition then has the given event and given Boolean evaluation as guard.

Another method would be adding a *requirement formula* subdomain in the semantic domain. The type corresponding to this domain would then consist of an event and a function of type `d -> Bool` (where `d` is the domain). An operator could then be introduced for 'applying' requirements to plants (as similarly done with restrictions). Applying a requirement to a plant would then come down to strengthening guards with the given function (using relational composition) of transitions with the given event.

Part IV

Discussion

Chapter 9

Conclusion

In this chapter we answer the research question introduced in the introduction. Recall that we defined the research question as follows.

- How can a functional programming language be used when developing tools for defining and simulating operational models with maintainability and extendibility taken into consideration?

In Chapter 6, we have introduced D-systems, which is our own formalism for defining discrete event systems based on the X-machine formalism by Samuel Eilenberg. This formalism overcomes some shortcomings of the EFA formalism (on which the CIF3 language is based on). We successfully implemented X-Control, which is our DSL for specifying discrete event systems, in a functional programming language (Haskell). The implementation of the semantic domain, which is directly based on our D-systems formalism, provides a minimalistic core for our toolchain. The type system and the declarative style of functional programming, allows for a semantic domain implementation that corresponds with the mathematical concepts and notations from Chapter 6. The implementation of our simulator for this core is then relatively straightforward. We have implemented X-Control as an internal DSL. The declarative style of Haskell and the `State` monad allowed us to quickly create an internal syntax, without too many restrictions (by the host language). The created valuation functions 'interpret' our syntax as elements of our semantic domain. As we have seen in Section 8.5, most of our proposed extensions can be realized by introducing new syntax elements, and modifying the valuation functions. This also shows the advantage of having a minimalistic implementation of the semantics, which captures the essence of our language.

Chapter 10

Further Work

On the toolchain side, we could implement the extensions discussed in Section 8.5. This means we could implement parameterization of systems, lists of systems, nested modules, event aliases, boolean expressions, and/or being able to define requirements based on formulae. We could also implement X-Control as an external DSL. Since an external DSL does not exist within the host language, we have more freedom in designing syntax constructs. This means we can create a more compact syntax, which could also be easier to use.

Suppose we have implemented this external syntax, we can then try to model the projects discussed in [20] and [21]. This would give us a way to compare the CIF3 language with the X-Control language.

Since our implementation of Algorithm 3 is basically a backtracking procedure, one might consider a more efficient implementation. In [18] an implementation based on the Binary Decision Diagram (BDD) data structure is suggested. BDDs allow for efficient evaluation and manipulation of Boolean expressions, which can aid in efficiently computing the predicates in the algorithm.

We could also do some further work on the theory discussed in Part II. We may be able to base our formalism on the XDI model described in [26]. An XDI specification (also called a process) is a triple (I, O, f), where I is the set of input symbols (comparable with the uncontrollable events from Section 5), O is the set of output symbols (comparable with the controllable events from Section 5), and f is the trace function which is of type $(I \cup O)^* \to \{\top, \nabla, \Box, \Delta, \bot\}$. Essentially f maps traces to one of the five results. $\top$ (top) means that the process has the obligation not to produce output symbols leading to this trace, ∇ (transient) means that the process is obligated to send an output, $\Box$ (quiescent) means that the process has no obligations, Δ (demanding) means that the process has the obligation to send some input symbol, and $\bot$ (bottom) means that the process fails due to an unexpected input symbol. Suppose we have some process P. The reflection of P, denoted as $\backsim P$, mirrors the results of the function f. This means that $f_P(t) = \top \iff f_{\backsim P}(t) = \bot$, $f_P(t) = \nabla \iff f_{\backsim P}(t) = \Delta$, and vice versa. We also have the ordering $\sqsubseteq$ on processes. Suppose $P \sqsubseteq P'$ (also denoted as P' refines P), then we say if $f_P(t) = \top$ for some $t \in (I \cup O^*)$ then $f_{P'}(t) = \top$. In [17] a composition operator $||$ is introduced for processes. This can be used to define the so called *design equation* which is defined as follows. Suppose we have some specification R (comparable with the requirements described in Chapter 6) and some given process P (which could be considered the plant), what are the processes C such that the following inequality is satisfied.

$$P \,||\, C \sqsupseteq R$$

In order words, which processes C can be composed with P such that the result refines R. Such a process C could be considered a 'supervisor' for P. The $\sqsubseteq$-least solution (the solution 'smallest' according to the ordering $\sqsubseteq$) is the so-called Galois connection, which is defined as follows.

$$\backsim(P \,||\, \backsim R)$$

This process could then considered the 'maximally permissive supervisor'. In [16], tools are discussed for finding this smallest solution. If we could define these concepts and notations ($\backsim$, $\sqsubseteq$, and the design equation) for our formalism in Chapter 6, then we may obtain less convoluted definitions regarding supervisory control synthesis (in Section 6.7).

Bibliography

[1] Cif 3. `http://cif.se.wtb.tue.nl`, 2016 (accessed July 27, 2020).

[2] Parsing expressions and statements. `https://wiki.haskell.org/Parsing_expressions_and_statements`, 2018 (accessed July 23, 2020).

[3] Glasgow haskell compiler. `https://www.haskell.org/ghc/`, 2020 (accessed August 1, 2020).

[4] J. Agron. Domain-specific language for hw/sw co-design for fpgas. In Walid Mohamed Taha, editor, *Domain-Specific Languages*, pages 262–284, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] G. Ali and N. A. Zafar. Modeling agent-based systems using x-machine and z notation. In *2010 Second International Conference on Communication Software and Networks*, pages 249–253, 2010.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, page 174–184, New York, NY, USA, 1998. Association for Computing Machinery.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. *SIGPLAN Not.*, 34(1):174–184, September 1998.

[8] M. D. Campos and L. S. Barbosa. Implementation of an orchestration language as a haskell domain specific language. *Electronic Notes in Theoretical Computer Science*, 255:45 – 64, 2009. Proceedings of the 8th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2009).

[9] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Springer Science+Business Media New York, New York, 1999.

[10] S. Eilenberg. *Automata, Languages, and Machines*, volume 59A. Academic press, Cambridge, Massachusetts, 1974.

[11] M. Erwig and E. Walkingshaw. Semantics-driven DSL design. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, (2011):56–80, 2012.

[12] M. Grabmüeller and D. Kleeblatt. Harpy: Run-time code generation in haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, page 94, New York, NY, USA, 2007. Association for Computing Machinery.

[13] R. M. Hierons and M. Harman. Testing conformance of a deterministic implementation against a non-deterministic stream x-machine. *Theoretical Computer Science*, 323(1):191 – 233, 2004.

[14] P. Kefalas, G. Eleftherakis, and E. Kehris. Communicating x-machines: a practical approach for formal and modular specification of large systems. *Information and Software Technology*, 45(5):269 – 280, 2003.

[15] A. M. Sloane M. Mernik, J. Heering. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[16] W. C. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. PhD thesis, Department of Computer Science, University of Groningen, The Netherlands, 2000.

[17] W. C. Mallon, J. T. Udding, and T. Verhoeff. Analysis and applications of the xdi model. In *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 231–242, 1999.

[18] L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson. Nonblocking and safe control of discrete-event systems modeled as extended finite automata. *IEEE Transactions on Automation Science and Engineering*, 8(3):560–569, 2011.

[19] B. Razet. Finite eilenberg machines. In Oscar H. Ibarra and Bala Ravikumar, editors, *Implementation and Applications of Automata*, pages 242–251, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[20] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak, and J. E. Rooda. Supervisory control synthesis for a waterway lock. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1562–1563, 2017.

[21] F. F. H. Reijnen, J. J. Verbakel, J. M. van de Mortel-Fronczak, and J. E. Rooda. Hardware-in-the-loop set-up for supervisory controllers with an application: the prinses marijke complex. In *2019 IEEE Conference on Control Technology and Applications (CCTA)*, pages 843–850, 2019.

[22] M. A. Reniers and J. M. van de Mortel-Fronczak. *Supervisory Control*, pages 57–125,135. 2019.

[23] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 42–51, 2014.

[24] M. Sköldstam, K. Åkesson, and M. Fabian. Supervisory control applied to automata extended with variables -revised supervisory control applied to automata extended with variables - revised. 01 2008.

[25] H. P. J. van Geldrop, J. C. S. P. van der Woude, and T. Verhoeff. *Declarative Programming (2IPH0) - Lecture Notes Part 2*. 2019.

[26] T. Verhoeff. Analyzing specifications for delay-insensitive circuits. In *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 172–183, 1998.

Appendices

Appendix A

Algebraic Properties of Synchronization

In this chapter, we discuss the properties *commutativity* and *associativity* of the synchronous product ($||$) operator on D-systems based on the equivalence relation discussed in Section 6.4. These properties are useful to have since they guarantee that the order in which the operator is applied on a set of D-systems does not matter. It should be noted that we have not been able to locate proofs for these properties for the synchronous product operator for EFA.

Conjecture ($||$) is commutative.

Proof sketch: Suppose we have D_1-system $((Q_1, I_1, T_1, \Phi_1, \delta_1), E_c^1, E_u^1, D_0^1)$ and D_2-system $((Q_2, I_2, T_2, \Phi_2, \delta_2), E_c^2, E_u^2, D_0^2)$. For brevity we say $E_i = E_c^i \cup E_u^i$. We have to prove that $S_1 \,||\, S_2 \backsimeq S_2 \,||\, S_1$. Suppose we have the path p

$$(q_0^1, q_0^2) \xrightarrow{t_1} \ldots \xrightarrow{t_n} (q_n^1, q_n^2)$$

with label ω in $S_1 \,||\, S_2$. We split p in sub paths: $p_1 \ldots p_m$ with labels $\omega_1, \ldots, \omega_m$. Paths $p_1, p_3 \ldots$ with labels are paths with events in E_2. Paths $p_2, p_4 \ldots$ have only events in E_1. In S_2 we have

$$q_0^2 \xrightarrow{\omega_1'}{}^{*} q_i^2 \xrightarrow{\omega_3'}{}^{*} q_j^2 \xrightarrow{\omega_5'}{}^{*} \ldots$$

where $\omega_k' \equiv \omega_k$. We can then observe that we have in $S_2 \,||\, S_1$

$$(q_0^2, q_0^1) \xrightarrow{\omega_1''} *(q_i^2, q_i^1) \xrightarrow{\omega_2''}{}^{*} (q_i^2, {q_i^1}') \xrightarrow{\omega_3''}{}^{*} (q_j^2, q_j^1) \xrightarrow{\omega_4''}{}^{*} (q_j^2, {q_j^1}') \xrightarrow{\omega_5''}{}^{*} \ldots$$

where $\omega_k'' \equiv \omega_k$. We can conclude that there is a path with label $\omega' = \omega_1'' \ldots \omega_m''$ in $S_2 \,||\, S_1$ with $\omega' \equiv \omega$.

The same reasoning can be implied for proving that if there is some path p in $S_2 \,||\, S_1$ with label ω, then there is some path p' in $S_1 \,||\, S_2$ with label ω' such that $\omega \equiv \omega'$. We can conclude that $S_1 \,||\, S_2 \backsimeq S_2 \,||\, S_1$.

Conjecture ($||$) is associative.

Proof sketch:

Suppose we have D_1-system $((Q_1, I_1, T_1, \Phi_1, \delta_1), E_c^1, E_u^1, D_0^1)$, D_2-system $((Q_2, I_2, T_2, \Phi_2, \delta_2), E_c^2, E_u^2, D_0^2)$ and D_3-system $((Q_3, I_3, T_3, \Phi_3, \delta_3), E_c^3, E_u^3, D_0^3)$. For brevity we say $E_i = E_c^i \cup E_u^i$. We have to prove that $S_1 \,||\, (S_2 \,||\, S_3) \backsimeq (S_1 \,||\, S_2) \,||\, S_3$. Suppose we have the path p

$$(q_0^1, (q_0^2, q_0^3)) \xrightarrow{t_1} \ldots \xrightarrow{t_n} (q_n^1, (q_n^2, q_n^3))$$

with label ω in $S_1 \,||\, (S_2 \,||\, S_3)$. We then split the path in sub paths $p_1 \ldots p_m$ with labels $\omega_1, \ldots, \omega_m$. Paths $p_1, p_3 \ldots$ have events in $E_2 \cup E_3$. $p_2, p_4, \ldots$ have events only in E_1. In $S_2 \,||\, S_3$ we have

$$(q_0^2, q_0^3) \xrightarrow{\omega_1^{D_2 \otimes D_3}}{}^{*} (q_1^2, q_1^3) \xrightarrow{\omega_3^{D_2 \otimes D_3}}{}^{*} (q_3^2, q_3^3) \ldots$$

Where $\omega_k^{D_2 \otimes D_3} \equiv \omega_k$. We split the sub paths into paths such that $p_1, p_5, \ldots$ have events in E_2 and $p_3, p_7, \ldots$ have events only in E_3. In S_2 we have

$$q_0^2 \xrightarrow{\omega_1^{D_2}}{}^{*} q_i^2 \xrightarrow{\omega_5^{D_2}}{}^{*} q_j^2 \ldots$$

where $\omega_k^{D_2} \equiv \omega_k$. In $S_1 \,||\, S_2$ we have

$$(q_0^1, q_0^2) \xrightarrow{\omega_1^{D_1 \otimes D_2}}{}^{*} (q_i^1, q_i^2) \xrightarrow{\omega_2^{D_1 \otimes D_2}}{}^{*} (q_i^{1\prime}, q_i^2) \xrightarrow{\omega_4^{D_1 \otimes D_2}}{}^{*} (q_i^{1\prime\prime}, q_i^2) \ldots$$

where $\omega_k^{D_1 \otimes D_2} \equiv \omega_k$. In $(S_1 \,||\, S_2) \,||\, S_3$ we have

$$((q_0^1, q_0^2), q_0^3) \xrightarrow{(D_1 \otimes D_2) \otimes D_3}{}^{*} ((q_i^1, q_i^2), q_i^3) \xrightarrow{(D_1 \otimes D_2) \otimes D_3}{}^{*} ((q_i^{1\prime}, q_i^2), q_i^3) \xrightarrow{(D_1 \otimes D_2) \otimes D_3}{}^{*} ((q_i^{1\prime}, q_i^2), q_i^{3\prime}) \ldots$$

where $\omega_k^{(D_1 \otimes D_2) \otimes D_3} \equiv \omega_k$. We can conclude that $\omega' = \omega_1^{(D_1 \otimes D_2) \otimes D_3} \ldots \omega_m^{(D_1 \otimes D_2) \otimes D_3}$ in $(D_1 \otimes D_2) \otimes D_3$ has $\omega' \equiv \omega$.

The same reasoning can be implied for proving that if there is some path p in $(S_1 \,||\, S_2) \,||\, S_3$ with label ω, then there is some path p' in $S_1 \,||\, (S_2 \,||\, S_3)$ with label ω' such that $\omega \equiv \omega'$. We can conclude that $S_1 \,||\, (S_2 \,||\, S_3) \backsimeq (S_1 \,||\, S_2) \,||\, S_3$.

Appendix B

Proof of Correctness Outline Supervisory Synthesis for D-Systems

In this chapter we present an outline of the proof of correctness for Algorithm 3. This proof outline is mostly based on the proof of correctness of the supervisory control syntheses algorithm for EFA (Algorithm 2) presented in [18].

Conjecture The algorithm terminates.

Proof sketch: Suppose we have some predicate $P : D \to \mathbb{B}$. There are only $|D|$ distinct predicates of type $P : D \to \mathbb{B}$. Since `FixPredicate` only expands the subdomain of D for which P_i is true, and the function terminates when P_i did not change after the last iteration, we can conclude that `FixPredicate` runs in $O(|D|)$ time. One iteration of the do-while loop takes $O(|D||C|)$ time. Line 8 will essentially 'block' certain $d \in D$ for each update in C. This can only be done $|D|$ times for each update. So the algorithms runs in $O(|D|^2|C|^2)$

Lemma Suppose the algorithm runs N iterations and for some $d \in D$ we have $B(d) = \mathsf{false}$, then there exist no $\omega \in (C_{N-1} \cup U)^*$ for which there exists $d' \in \rho_\omega(d_0)$ for which $B(d') = \mathsf{true}$.

Proof sketch: We prove the claim by induction on the length of the sequence of ω (denoted by $|\omega|$).

Base case: $|\omega| = 0$: If $|\omega| = 0$ then $\rho_\omega = id_D$, which implies $\rho_\omega(d)$.

Step case: $|\omega| > 0$: We assume that $\omega = \omega'\phi$ and that no $d' \in \omega'(d)$ has $B(d') = \mathsf{true}$ (IH). Suppose that $d'' \in \rho_\omega(d)$ has $B(d'') = \mathsf{true}$. Then there exists $d' \in \rho_{\omega'}(d)$ for which $d'' \in \rho_\phi(d')$. If $\phi \in C_{N-1}$ then $C_{N-1} \neq C_N$ because of line 8, which leads to a contraction. If $\phi \in U$ then $B(d')$ set to true in line 16, which contradicts with the induction hypothesis. So we can conclude that all $d'' \in \rho_\omega(d)$ have $B(d'') = \mathsf{false}$, which proves our claim.

Conjecture C_{N-1} and U are controllable with respect to C and U.

Proof sketch: Since we use the same set of uncontrollable updates, it is trivial to observe that C_{N-1} and U are controllable with respect to C and U.

Conjecture $(C_{N-1}, U) \preceq (C, U)$.

From line 8 it is trivial to observe that $(C_{N-1}, U) \preceq (C, U)$.

Conjecture C_{N-1} and U are non-blocking.

Proof sketch: Suppose $d \in D$ is non-blocking with respect to C_{N-1} and U. Then from lines 5 and 6 we can infer that $N(d) = \mathsf{true}$. Suppose $B(d) = \mathsf{false}$, then, according to our lemma, there exist no $\omega \in (C_{N-1} \cup U)^*$ such that there is $d' \in \rho_\omega$ with $B(d') = \mathsf{true}$, which implies for all $\omega \in (C_{N-1} \cup U)^*$ for all $d' \in \rho_\omega$ we have $N(d') = \mathsf{true}$. Suppose $B(d) = \mathsf{true}$. Assume there exists $\omega \in (C_{N-1} \cup U)^*$ with $\omega \notin U^*$, such that there exists blocking $d' \in \rho_\omega(d)$. Suppose $\omega = \omega'\phi\omega''$ where $\phi \in C_{N-1}$, $\omega'' \in U^*$, $d_1 \in \rho_{\omega'}(d)$, $d_2 \in \rho_\phi(d_1)$ and $d' \in \rho_{\omega''}(d_2)$. Since $\omega'' \in U^*$ and d' is blocking, then, inferring from line 7, we know that $B(d_3) = \mathsf{true}$, this contradicts with line 8 since $b_3 \in \rho_\phi(d_2)$ and $\phi \in C_{N-1}$. We can conclude that $\omega \in U^*$, which proves our claim.

Conjecture C_{N-1} and U are a maximally permissive.

Proof sketch: Suppose we have C' and U' which are a proper supervisor for C and U. Suppose we have completely non-blocking $d, d' \in D$ for which there exists $\omega' \in (C' \cup U')^*$ with $d' \in \rho_{\omega'}(d)$ and $B(d) = \mathsf{false}$. Suppose there is no $\omega \in (C_{N-1} \cup U)^*$ for which $d' \in \rho_\omega(d)$. There must exist $\omega' = \omega_1'\phi'\omega_2'$. Where $d_1 \in \rho_{\omega_1'}(d)$, $d_2 \in \rho_{\phi'}(d_1)$, $d' \in \rho_{\omega_2'}(d_2)$, and there exist $\omega_1 \in (C_{N-1} \cup U)^*$ with $d_1 \in \rho_{\omega_1'}(d)$ with no $\phi \in (C_{N-1} \cup U)$ with $d_2 \in \rho_\phi(d_1)$. Then we know $\phi' \notin U$. Because of line 8 we also know that $B(d_2) = \mathsf{true}$. Since $d' \in \rho_{\omega_2'}(d_2)$ we know that $N(d_2) = \mathsf{true}$ at one point which implies that $B(d_2) = \mathsf{false}$ at one point. This means that at some point $B(d_2)$ is set to true. This is done in line 7 when there is some sequence $\omega'' \in U^*$ for which there exists $d'' \in \rho_{\omega''}(d_2)$ where at no point $N(d'')$ is set to true. Since $\omega'' \in U^*$ we know that $\omega_1'\phi'\omega'' \in (C'' \cup U)^*$ with $d'' \in \rho_{(\omega_1'\phi'\omega'')}(d)$, which contradicts with the fact that C'' is non-blocking.

Appendix C

Semantic Domain Implementation

In this appendix, the implementation of the semantic domain of X-Control is discussed in more detail.

[3]:
```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TupleSections #-}
import Data.Monoid
import Data.List
import Data.Maybe
import Data.Foldable
import Data.Semigroup
import Control.Monad
```

```
Line 5: Unused LANGUAGE pragma
Found:
{-# LANGUAGE TupleSections #-}
Why not:
```

1 Automata

We first model a free monoid with a arbitrary base as snoc list structure. We use a snoc list structure instead of the regular cons list, since appending symbols at the end of a snoc list is more convenient. This is useful for generating traces.

[4]:
```
data FreeMonoid a = Empty | FreeMonoid a :> a

single :: a -> FreeMonoid a
single x = Empty :> x

fromList :: [a] -> FreeMonoid a
fromList = foldr (flip (:>)) Empty

instance Semigroup (FreeMonoid a) where
    fm <> Empty = fm
    fm1 <> (fm2 :> x) = (fm1 <> fm2) :> x

instance Monoid (FreeMonoid a) where
```

```haskell
    mempty = Empty

instance Foldable FreeMonoid where
    foldMap f Empty = mempty
    foldMap f (fm :> x) = foldMap f fm <> f x

instance (Show a) => Show (FreeMonoid a) where
    show = show . toList
```

We define a class for a *type* for an automaton. Defining such a class allows us to add constraints later when needed.

[5]:
```haskell
class AutomatonType a
```

An example of an automaton type is `Char`.

[6]:
```haskell
instance AutomatonType Char
```

We define a Σ-automaton (Q, I, T, δ) as follows.

[7]:
```haskell
data Automaton a b
  where
    Automaton
        :: (AutomatonType a
           , Eq b
           )
        => [b]              -- states
        -> [b]              -- initial states
        -> [b]              -- terminal states
        -> [a]              -- alphabet
        -> (b -> a -> [b])  -- transition relation
        -> Automaton a b

states :: Automaton a b -> [b]
states (Automaton qs is ts ss (==>)) = qs

initialStates :: Automaton a b -> [b]
initialStates (Automaton qs is ts ss (==>)) = is

terminalStates :: Automaton a b -> [b]
terminalStates (Automaton qs is ts ss (==>)) = ts

alphabet :: Automaton a b -> [a]
alphabet (Automaton qs is ts ss (==>)) = ss

delta :: Automaton a b -> (b -> a -> [b])
delta (Automaton qs is ts ss (==>)) = (==>)
```

```haskell
instance (Show a, Show b) => Show (Automaton a b)
  where
    show (Automaton qs is ts alph (==>))
      =
        "States: " ++ show qs ++ "\n" ++
        "Initial States: " ++ show is ++ "\n" ++
        "Terminal States: " ++ show ts ++ "\n" ++
        "Alphabet: [\n" ++ prettyAlphabet ++ "]\n\n" ++
        "Transitions:\n" ++
        transitions
      where
        transitions = do q <- qs
                         label <- alph
                         let qs' = q ==> label
                         guard $ not (null qs')
                         let shortLabel | length (show label) > 40 = take 40␣
↪(show label) ++ "..."
                                        | otherwise                 = show label
                         "" ++ show q ++ " ==> " ++ shortLabel ++ " = " ++ show␣
↪qs' ++ "\n\n"
        prettyAlphabet = do a <- alph
                            let label | length (show a) > 40 = take 40 (show a)␣
↪++ "..."
                                      | otherwise            = show a
                            "    " ++ label ++ ",\n"
```

We use GDT to enforce the type classes. `a` is the type of the symbols on the transitions, which should be of the type automataType. `b` is the type of the state values, which should be an instance of `Eq`, since we need to compare two states for equality. Since we want to be as faithfull as possible to the algebraic automata theory discussed in chapter 3, we model the transitions relation δ as a function from an instance of the state type to a list of instances of the state type.

The type of Σ (and Σ^*) is `a` which should be an instance of `AutomatonType` and the type of Q, I and T is `b` which should be an instance of `Eq` (since one must be able to check if a state is in I or T).

An example automaton is constructed as follows.

[8]:
```haskell
automaton1 :: Automaton Char Int
automaton1 = Automaton qs is ts ss (==>)
  where
    qs = [0, 1, 2]
    is = [0]
    ts = [0, 1]
    ss = ['a', 'b']
    0 ==> 'a' = [1]
    0 ==> 'b' = [2]
    1 ==> 'a' = [2]
```

```
    1 ==> 'b' = [0]
    2 ==> 'a' = [2]
    2 ==> 'b' = [2]
```

We can define the function γ as follows.

[9]:
```
gamma :: AutomatonType a => Automaton a b -> [b] -> FreeMonoid a -> [b]
gamma automaton states word = do state <- states
                                 foldlM (delta automaton) state word
```

We can use γ to determine whether $\omega \in \mathcal{L}(A)$ (recall that $\mathcal{L}(A) = \{\omega \in \Sigma^* | I\omega \cap T \neq \emptyset\}$).

[10]:
```
checkAccept :: (AutomatonType a, Eq b) => Automaton a b -> FreeMonoid a -> Bool
checkAccept automaton word = intersect (is ==>* word) ts /= [] where
    (==>*) = gamma automaton
    is = initialStates automaton
    ts = terminalStates automaton
```

A couple of example strings:

[11]:
```
"empty in L(A)?"
checkAccept automaton1 Empty
"a in L(A)?"
checkAccept automaton1 $ single 'a'
"ababa in L(A)?"
checkAccept automaton1 $ single 'a' <> single 'b' <> single 'a' <> single 'b'␣
 ↪<> single 'a'  --single"ababa"
"ababb in L(A)?"
checkAccept automaton1 $ single 'a' <> single 'b' <> single 'a' <> single 'b'␣
 ↪<> single 'b'  --"ababb"
"abaaba in L(A)?"
checkAccept automaton1 $ single 'a' <> single 'b' <> single 'a' <> single 'a'␣
 ↪<> single 'b' <> single 'a'  --"abaababab"
```

```
"empty in L(A)?"

True

"a in L(A)?"

True

"ababa in L(A)?"

True
```

```
"ababb in L(A)?"
```

```
False
```

```
"abaaba in L(A)?"
```

```
False
```

We can define a function for obtaining the language of some automaton. We first define a function which, given a state trace tuple, computes a list of possible successor state trace tuple according to the transitions relation.

[12]:
```
getNextStates :: AutomatonType a => Automaton a b -> [a] -> (b, FreeMonoid a)␣
↪-> [(b, FreeMonoid a)]
getNextStates automaton symbols (state, trace) = [(nextState, trace :> sigma)
                                                 | sigma <- symbols
                                                 , nextState <- state ==> sigma
                                                 ]
  where
    (==>) = delta automaton

getAllNextStates :: AutomatonType a => Automaton a b -> (b, FreeMonoid a) ->␣
↪[(b, FreeMonoid a)]
getAllNextStates automaton = getNextStates automaton (alphabet automaton)
```

With getLanguage we can then obtain a list of successful traces of some automaton. Note that this list can be infinite. In that case a subset of the language can be retrieved due to lazy evaluation.

[13]:
```
getLanguage :: Automaton a b -> [(b, FreeMonoid a)]
getLanguage automaton@(Automaton qs is ts alph (==>)) = step [(state, Empty) |␣
↪state <- is]
  where
    step [] = []
    step options = solutions ++ step rest
      where
        solutions = filter ((`elem` ts) . fst) options
        rest = concatMap (getAllNextStates automaton) options
```

As an example we take elements from the language of our example automaton:

[14]:
```
print $ take 5 $ getLanguage automaton1
```

```
[(0,""),(1,"a"),(0,"ab"),(1,"aba"),(0,"abab")]
```

2 X-Machines

In X-machines, the labels on the transitions (the labels of the alphabet), are binary relations on some domain X. We will now model this concept of binary relations. Our model for binary relations on domain `a` has a label, which will be used to test relations on equivalence, and a function with type `a -> [a]`. Given some instance x of `a`, the function returns all instances y of `a` to which x relates.

[15]:
```
data BinaryRel a = BinaryRel String (a -> [a])

identityRel :: BinaryRel a
identityRel = BinaryRel "id" (:[])

instance AutomatonType (BinaryRel a)

instance Show (BinaryRel a)
  where
    show (BinaryRel label rel) = label

instance Eq (BinaryRel a)
  where
    (BinaryRel label1 rel1) == (BinaryRel label2 rel2) = label1 == label2

instance Semigroup (BinaryRel a)
  where
    (BinaryRel label1 rel1) <> (BinaryRel label2 rel2) = BinaryRel (label1 ++ ";
 ↪" ++ label2) (rel1 >=> rel2)

instance Monoid (BinaryRel a) where
    mempty = identityRel

getRelation :: BinaryRel a -> (a -> [a])
getRelation (BinaryRel label rel) = rel

getLabel :: BinaryRel a -> String
getLabel (BinaryRel label rel) = label
```

Making `BinaryRel` an instance of `Monoid` allows for composition of relations.

Recall that an X-machine is just an automaton where the alphabet was a set of binary relations on some domain X. Based on this concept, we have the following definition of X-machine.

[16]:
```
type Machine a b = Automaton (BinaryRel a) b
```

As an example we define a machine which computes the factorial of some arbitrary number n.

[17]:
```
machine1 :: Machine (Int, Int) Int
machine1 = Automaton qs is ts tp delta where
    mul = BinaryRel "mul" (\(n, m) -> [(n, n * m)])
```

```haskell
    minus = BinaryRel "minus" (\(n, m) -> [(n - 1, m)])
    checkZero = BinaryRel "checkZero" r
        where
            r (0, m) = [(0,m)]
            r _      = []

    qs = [0, 1, 2]
    is = [0]
    ts = [2]
    tp = [mul, minus, checkZero]
    delta q (BinaryRel label rel) = q ==> label

    0 ==> "mul" = [1]
    0 ==> "checkZero" = [2]
    1 ==> "minus" = [0]
    _ ==> _ = []
```

[18]:
```haskell
print $ take 10 $ getLanguage machine1
```

```
[(2,[checkZero]),(2,[mul,minus,checkZero]),(2,[mul,minus,mul,minus,checkZero]),(2,[mul,minus,mu
```

Using the language of the machine we can (inefficiently) compute the *characteristic relation* of a machine.

[19]:
```haskell
charRelNaive :: Automaton (BinaryRel a) b -> String -> BinaryRel a
charRelNaive aut l = BinaryRel l (\x -> concatMap (\(st, label) -> getRelation␣
 ↪(fold label) x) (getLanguage aut))
```

[20]:
```haskell
head (getRelation (charRelNaive machine1 "m1") (25,1))
```

```
(0,7034535277573963776)
```

In the next example we create a machine which checks, given a string of a's and b's, if the number of a's is equal to the number of b's in the string.

[21]:
```haskell
machine2 :: Automaton (BinaryRel (String, Int)) String
machine2 = Automaton qs is ts ss delta where
    zero =  BinaryRel "zero" (\(str, n) -> [(str, n) | n == 0])
    greater = BinaryRel "greater" (\(str, n) -> [(str, n) | n > 0])
    lesser = BinaryRel "lesser" (\(str, n) -> [(str, n) | n < 0])
    plus = BinaryRel "plus" r
        where
            r ('a':str,n) = [(str, n + 1)]
            r _ = []
    minus = BinaryRel "min" r
        where
            r ('b':str, n) = [(str, n - 1)]
```

```
            r _ = []

    qs = ["0", "<", ">"]
    is = ["0"]
    ts = ["0"]
    ss = [plus, minus, plus <> zero, minus <> zero, plus <> lesser, minus <>␣
↪greater]
    delta q (BinaryRel label rel) = q ==> label

    "0" ==> "min" = ["<"]
    "0" ==> "plus" = [">"]
    "<" ==> "min" = ["<"]
    "<" ==> "plus;lesser" = ["<"]
    "<" ==> "plus;zero" = ["0"]
    ">" ==> "plus" = [">"]
    ">" ==> "min;greater" = [">"]
    ">" ==> "min;zero" = ["0"]
    _ ==> _ = []
```

We now define a more efficient way to compute the *characteristic relation*.

[22]:

```
getNextSteps :: AutomatonType a => Automaton a b -> [a] -> b -> [(b, a)]
getNextSteps automaton symbols state = do sigma <- symbols
                                          endState <- state ==> sigma
                                          return (endState, sigma)
    where
        (==>) = delta automaton

getAllNextSteps :: AutomatonType a => Automaton a b -> b -> [(b, a)]
getAllNextSteps automaton = getNextSteps automaton (alphabet automaton)

charRel :: Automaton (BinaryRel a) b -> String -> BinaryRel a
charRel automaton@(Automaton qs is ts alph (==>)) l =  BinaryRel l rel
    where
        rel x = do (st', x', tr) <- step [(state, x, Empty) | state <- is]
                   return x'

        step [] = []
        step options = solutions ++ step rest
            where
                solutions = filter (\(st, val, tr) -> st `elem` ts) options
                rest = do (st, x, tr) <- options
                          (st', rel) <- getAllNextSteps automaton st
                          x' <- getRelation rel x
                          return (st', x', tr :> rel)
```

[23]:

```
print $ take 10 (getRelation (charRel machine2 "m2") ("bbbaabaaba",0))
```

```
[("bbbaabaaba",0),("ba",0),("",0)]
```

3 Systems

We now implement the *D-system* formalism discussed in chapter 6. We first define a type for events and the tuples (ϕ_1, e, ϕ_2).

[24]:
```
type Event a = BinaryRel a

type EventUpdate a = (BinaryRel a, Event a, BinaryRel a)
getEventUpdateRel :: EventUpdate a -> (a -> [a])
getEventUpdateRel (rel1, e, rel2) = getRelation (rel1 <> e <> rel2)

instance AutomatonType (EventUpdate a)
```

We then define our `EventMachine` type, which is a automaton with alphabet of type `EventUpdate`. We also define a `StateLabel` datatype, of which instances can be composed in a tree like structure. This helps us in defining the synchronous product operatior which we discuss later.

[25]:
```
data StateLabel = SingleLabel String | JointLabel StateLabel StateLabel
getStateLabel :: StateLabel -> String
getStateLabel (SingleLabel str) = str
getStateLabel l = "(" ++ rec l ++ ")"
  where
    rec (SingleLabel str) = str
    rec (JointLabel l1 l2) = rec l1 ++ "," ++ rec l2

instance Show StateLabel
    where
        show = getStateLabel

instance Eq StateLabel
    where
        sl1 == sl2 = getStateLabel sl1 == getStateLabel sl2

type EventMachine a = Automaton (EventUpdate a) StateLabel
```

The controllable and uncontrollable events, the initial values and the *EventMachine* will be the components of the `System` datatype. `System` will be used to model discrete event systems.

[26]:
```
data System a = System
    { machine :: EventMachine a
    , controllableEvents :: [BinaryRel a]
    , uncontrollableEvents :: [BinaryRel a]
    , domain :: [a]
    , initialValues :: [a]
    }
```

```haskell
instance (Show a) => Show (System a)
    where
      show (System machine contr uncontr dom initval) =
          "Controllable Events: " ++ show contr ++
          "\nUncontrollable Events: " ++ show uncontr ++
          "\nDomain: " ++ prettyDomain ++
          "\nInitial values: " ++ show initval ++
          "\n\nMachine:\n" ++
          show machine
        where
          prettyDomain | length (show dom) > 100 = take 100 (show dom) ++ "..."
                       | otherwise                = show dom
```

We now discuss some examples. In the first example we model a simple sensor.

[27]:
```haskell
sensorSwitchEvent = BinaryRel "sensorSwitch" rel
            where
                rel b = [not b]

sensorSystem :: System Bool
sensorSystem = System sensorMachine [] [sensorSwitchEvent] [False, True] [False]
    where
        sensorMachine = Automaton qs is ts phis delta
            where
                offState = SingleLabel "sensorOff"
                onState = SingleLabel "sensorOn"

                qs = [offState, onState]
                is = [offState]
                ts = [offState]
                phis = [(identityRel, sensorSwitchEvent, identityRel)]
                delta q (phi, ev, phi') = getStateLabel q ==> getLabel ev

                "sensorOff" ==> "sensorSwitch" = [onState]
                "sensorOn"  ==> "sensorSwitch" = [offState]
```

In this example we model a simple actuator.

[28]:
```haskell
actuatorSwitchEvent = BinaryRel "actuatorSwitch" rel
            where
                rel b = [not b]

actuatorSystem :: System Bool
actuatorSystem = System actuatorMachine [actuatorSwitchEvent] [] [False, True]␣
 ↪[False]
    where
```

```
        actuatorMachine = Automaton qs is ts phis delta
            where
                offState = SingleLabel "actuatorOff"
                onState = SingleLabel "actuatorOn"

                qs = [offState, onState]
                is = [offState]
                ts = [offState]
                phis = [(identityRel, actuatorSwitchEvent, identityRel)]
                delta q (phi, ev, phi') = getStateLabel q ==> getLabel ev

                "actuatorOff" ==> "actuatorSwitch" = [onState]
                "actuatorOn"  ==> "actuatorSwitch" = [offState]
```

Requirements can be modeled by machines. In this example we model the requirement 'the actuator may only switch on/off after the sensor has been switched on/off'.

[29]:
```
simpleRequirement1 :: System (Bool, Bool)
simpleRequirement1 = System requirementMachine [actuatorSwitchEvent]␣
 ↪[sensorSwitchEvent] dom [(False, False)]
    where
        dom = [(b1, b2) | b1 <- [False, True], b2 <- [False, True]]

        sensorSwitchEvent = BinaryRel "sensorSwitch" rel
            where
                rel (bs, ba) = [(not bs, ba)]

        actuatorSwitchEvent = BinaryRel "actuatorSwitch" rel
            where
                rel (bs, ba) = [(bs, not ba)]

        requirementMachine = Automaton qs is ts phis delta
            where
                r0 = SingleLabel "r0"
                r1 = SingleLabel "r1"

                qs = [r0, r1]
                is = [r0]
                ts = [r0]
                phis = [(identityRel, sensorSwitchEvent, identityRel),
                        (identityRel, actuatorSwitchEvent, identityRel)]
                delta q (phi, ev, phi') = getStateLabel q ==> getLabel ev

                "r0" ==> "sensorSwitch" = [r1]
                "r1" ==> "sensorSwitch" = [r1]
                "r1" ==> "actuatorSwitch" = [r0]
```

In the next example we model the requirement 'the actuator may only switch on/off after the sensor has been switched on.

[30]:
```
simpleRequirement2 :: System (Bool, Bool)
simpleRequirement2 = System requirementMachine [actuatorSwitchEvent]␣
 ↪[sensorSwitchEvent] dom [(False, False)]
    where
        dom = [(b1, b2) | b1 <- [False, True], b2 <- [False, True]]

        sensorSwitchEvent = BinaryRel "sensorSwitch" rel
            where
                rel (bs, ba) = [(not bs, ba)]

        actuatorSwitchEvent = BinaryRel "actuatorSwitch" rel
            where
                rel (bs, ba) = [(bs, not ba)]

        requirementMachine = Automaton qs is ts phis delta
            where
                i1 = BinaryRel "i1" rel
                    where
                        rel (False, ba) = [(False, ba)]
                        rel _ = []

                i2 = BinaryRel "i2" rel
                    where
                        rel (True, ba) = [(True, ba)]
                        rel _ = []

                r0 = SingleLabel "r0"
                r1 = SingleLabel "r1"


                qs = [r0, r1]
                is = [r0]
                ts = [r0]
                phis = [(identityRel, sensorSwitchEvent, identityRel),
                        (identityRel, actuatorSwitchEvent, identityRel),
                        (identityRel, sensorSwitchEvent, i1),
                        (identityRel, sensorSwitchEvent, i2)]
                delta q t = getStateLabel q ==> show t


                "r0" ==> "(id,sensorSwitch,i1)" = [r0]
                "r0" ==> "(id,sensorSwitch,i2)" = [r1]
                "r1" ==> "(id,sensorSwitch,id)" = [r1]
                "r1" ==> "(id,actuatorSwitch,id)" = [r0]
                _    ==> _ = []
```

For event systems we define a function for computing the possible traces of events which can occur in the system, together with the corresponding final domain values.

[31]:
```
getTraces :: System a -> [(FreeMonoid (BinaryRel a), a)]
getTraces eventSys = map (\(state', x', tr) -> (tr, x')) $ step [(state,␣
↪initVal, Empty)
                                                                | state <- is
                                                                , initVal <-␣
↪initVals
                                                                ]
    where
        mach@(Automaton qs is ts alph (==>)) = machine eventSys
        initVals = initialValues eventSys

        step [] = []
        step options = solutions ++ step rest
            where
                solutions = filter (\(st, val, tr) -> st `elem` ts) options
                rest = do (st, x, tr) <- options
                          (st', (phi, e, phi')) <- getAllNextSteps mach st
                          x' <- getRelation (phi <> e <> phi') x
                          return (st', x', tr :> e)
```

We take 3 traces of the sensor system for illustration.

[32]:
```
print $ take 3 $ getTraces sensorSystem
```

```
[([],False),([sensorSwitch,sensorSwitch],False),([sensorSwitch,sensorSwitch,sensorSwitch,senso
```

3.1 Synchronization

We now introduce the synchronous product operator for *EventSystems*. To do this we will do the following:

- Implement a data type to describe the composition of two domains.
- Implement projection and synchronization for *EventUpdates*.
- Implement synchronization for *EventSystems*.

When synchronizing two *EventSystems*, the domains of the two systems are composed in a certain way. This mostly comes down to having shared and non-shared parts of the domain, where the shared parts must be synchronized. To describe such a domain composition we define the data type `DomainComposition`.

[33]:
```
data DomainComposition d1 d2 dc = DomainComposition
    { combine :: d1 -> d2 -> dc
    , decompose :: dc -> (d1, d2)
    , checkComp :: d1 -> d2 -> Bool
    , extract1 :: dc -> d1
```

```
    , extract2 :: dc -> d2
    , augment1 :: dc -> d1 -> dc
    , augment2 :: dc -> d2 -> dc
    }
```

The fields of a `domainComposition` can be described as follows.

- `combine`: A mapping from instances of the two original domains to an instance of the composed domain (according to the composition)
- `decompose`: A mapping from an instance of the composed domain to the instances of the original domains.
- `checkComb`: Check if the instances of the two original domains can be mapped to the composed domain (in most cases, this would be checking of the shared domain values are equal).
- `extract1`: A mapping from an instance of the composed domain to the instance of the first original domain.
- `extract2`: A mapping from an instance of the composed domain to the instance of the second original domain.
- `augment1`: Suppose we have a value `vc` of the composed domain corresponding to the values `v1` of the first domain and `v2` of the second domain, and a value `v1'` of the first domain. `augment1` maps `vc` and `v1'` to the instance of the composed domain corresponding with `v1'` and `v2`.
- `augment2`: Suppose we have a value `vc` of the composed domain corresponding to the values `v1` of the first domain and `v2` of the second domain, and a value `v2'` of the second domain. `augment2` maps `vc` and `v2'` to the instance of the composed domain corresponding with `v1` and `v2'`.

We now define some example *DomainCompositions*. We first define a disjoint domain composition, where the two given domains are independent.

[34]:
```
disjointComposition :: DomainComposition d1 d2 (d1, d2)
disjointComposition = DomainComposition
    { combine = \v1 v2 -> (v1,v2)
    , decompose = id
    , checkComp = \_ _ -> True
    , extract1 = fst
    , extract2 = snd
    , augment1 = \(v1, v2) v1' -> (v1', v2)
    , augment2 = \(v1, v2) v2' -> (v1, v2')
    }
```

Next we define a joint domain composition. In this composition the two given domains are the same, and there values should be equal.

[35]:
```
jointComposition :: Eq d => DomainComposition d d d
jointComposition = DomainComposition
    { combine = const
    , decompose = \v -> (v, v)
    , checkComp = (==)
    , extract1 = id
```

```
    , extract2 = id
    , augment1 = \v v' -> v'
    , augment2 = \v v' -> v'
    }
```

Using a domain composition we can take binary relations on either `d1` or `d2`, and change their respective domains to `dc`.

[36]:
```
transformDomain :: (dc -> d) -> (dc -> d -> dc) -> BinaryRel d -> BinaryRel dc
transformDomain extract augment (BinaryRel l r) = BinaryRel l (\vc -> map␣
↪(augment vc) $ (r . extract) vc)

projectLeft :: DomainComposition d1 d2 dc -> BinaryRel d1 -> BinaryRel dc
projectLeft domComp = transformDomain (extract1 domComp) (augment1 domComp)

projectRight :: DomainComposition d1 d2 dc -> BinaryRel d2 -> BinaryRel dc
projectRight domComp = transformDomain (extract2 domComp) (augment2 domComp)
```

We also define a function parallelize a relation on `d1` and `d2` which results to a relation on `dc`.

[37]:
```
syncRel :: DomainComposition d1 d2 dc -> BinaryRel d1 -> BinaryRel d2 ->␣
↪BinaryRel dc
syncRel domComp (BinaryRel l1 r1) (BinaryRel l2 r2) = BinaryRel syncLabel␣
↪syncRel
    where
        syncLabel | l1 == l2  = l1
                  | otherwise = l1 ++ "||" ++ l2
        syncRel vc = do let (v1, v2) = decompose domComp vc
                        v1' <- r1 v1
                        v2' <- r2 v2
                        guard $ checkComp domComp v1' v2'
                        return (combine domComp v1' v2')
```

Using the aforementioned operations on binary relations, we define operation for synchronizing two event systems based on their events given a domain composition.

[38]:
```
getEventIntersection :: [BinaryRel a] -> [BinaryRel b] -> [(BinaryRel a,␣
↪BinaryRel b)]
getEventIntersection events1 events2 = [(ev1, ev2)
                                       | ev1 <- events1, ev2 <- events2
                                       , getLabel ev1  == getLabel ev2
                                       ]

getEventDifference :: [BinaryRel a] -> [BinaryRel b] -> [BinaryRel a]
getEventDifference events1 events2 = [ ev
                                     | ev <- events1,
                                       not $ any (\ev2 -> getLabel ev2 ==␣
↪getLabel ev) events2]
```

```haskell
syncEventSystems :: DomainComposition d1 d2 dc -> System d1 -> System d2 ->␣
↪System dc
syncEventSystems
    domainComp
    (System (Automaton qs1 is1 ts1 phis1 delta1) contr1 uncontr1 dom1 initVals1)
    (System (Automaton qs2 is2 ts2 phis2 delta2) contr2 uncontr2 dom2 initVals2)
  =
    System machine contr uncontr dom initVals
  where
    (<||>) = syncRel domainComp
    projectLeft' = projectLeft domainComp
    projectRight' = projectRight domainComp

    -- events from both systems
    eventsInterContr = getEventIntersection contr1 contr2
    eventsInterUncontr = getEventIntersection uncontr1 uncontr2
    -- events only in the first system
    eventsDiffContr1 = getEventDifference contr1 contr2
    eventsDiffUncontr1 = getEventDifference uncontr1 uncontr2
    -- events only in the second system
    eventsDiffContr2 = getEventDifference contr2 contr1
    eventsDiffUncontr2 = getEventDifference uncontr2 uncontr1

    dom = [combine domainComp d1 d2 | d1 <- dom1, d2 <- dom2, checkComp␣
↪domainComp d1 d2]
    initVals = zipWith (combine domainComp) initVals1 initVals2

    contr = map (uncurry (<||>)) eventsInterContr
            ++
            map projectLeft' eventsDiffContr1
            ++
            map projectRight' eventsDiffContr2

    uncontr = map (uncurry (<||>)) eventsInterUncontr
              ++
              map projectLeft' eventsDiffUncontr1
              ++
              map projectRight' eventsDiffUncontr2

    machine = Automaton qs is ts phis delta

    phisInter = [ ((phi1 <||> phi2, ev1 <||> ev2, phi1' <||> phi2'), t1, t2)
                | (ev1, ev2) <- eventsInterContr ++ eventsInterUncontr
                , t1@(phi1, ev1', phi1') <- phis1
                , ev1 == ev1'
                , t2@(phi2, ev2', phi2') <- phis2
```

```
               , ev2' == ev2
               ]
    phisDif1 = [ ((projectLeft' phi1, projectLeft' ev, projectLeft' phi1'), t)
               | ev <- eventsDiffContr1 ++ eventsDiffUncontr1
               , t@(phi1, ev', phi1') <- phis1
               , ev == ev'
               ]
    phisDif2 = [ ((projectRight' phi2, projectRight' ev, projectRight' phi2'),␣
↪t)
               | ev <- eventsDiffContr2 ++ eventsDiffUncontr2
               , t@(phi2, ev', phi2') <- phis2
               , ev == ev'
               ]

    qs = [ JointLabel q1 q2 | q1 <- qs1, q2 <- qs2]
    is = [ JointLabel i1 i2 | i1 <- is1, i2 <- is2]
    ts = [ JointLabel t1 t2 | t1 <- ts1, t2 <- ts2]
    phis = [evComb | (evComb, ev1, ev2) <- phisInter]
           ++
           [evComb | (evComb, ev) <- phisDif1]
           ++
           [evComb | (evComb, ev) <- phisDif2]
    delta (JointLabel q1 q2) phi = [ JointLabel q1' q2'
                                   | (phi', phi1, phi2) <- phisInter
                                   , phi' == phi
                                   , q1' <- delta1 q1 phi1
                                   , q2' <- delta2 q2 phi2
                                   ]
                                   ++
                                   [ JointLabel q1' q2
                                   | (phi', phi_1) <- phisDif1
                                   , phi' == phi
                                   , q1' <- delta1 q1 phi_1
                                   ]
                                   ++
                                   [ JointLabel q1 q2'
                                   | (phi', phi_2) <- phisDif2
                                   , phi' == phi
                                   , q2' <- delta2 q2 phi_2
                                   ]
```

We construct our plant by synchronizing the sensor and the actuator.

[39]:
```
plant :: System (Bool, Bool)
plant = syncEventSystems disjointComposition sensorSystem actuatorSystem

putStr $ show plant
```

```
Controllable Events: [actuatorSwitch]
Uncontrollable Events: [sensorSwitch]
Domain: [(False,False),(False,True),(True,False),(True,True)]
Initial values: [(False,False)]

Machine:
States: [(sensorOff,actuatorOff),(sensorOff,actuatorOn),(sensorOn,actuatorOff),(sensorOn,actua
Initial States: [(sensorOff,actuatorOff)]
Terminal States: [(sensorOff,actuatorOff)]
Alphabet: [
    (id,sensorSwitch,id),
    (id,actuatorSwitch,id),
]

Transitions:
(sensorOff,actuatorOff) ==> (id,sensorSwitch,id) = [(sensorOn,actuatorOff)]

(sensorOff,actuatorOff) ==> (id,actuatorSwitch,id) = [(sensorOff,actuatorOn)]

(sensorOff,actuatorOn) ==> (id,sensorSwitch,id) = [(sensorOn,actuatorOn)]

(sensorOff,actuatorOn) ==> (id,actuatorSwitch,id) = [(sensorOff,actuatorOff)]

(sensorOn,actuatorOff) ==> (id,sensorSwitch,id) = [(sensorOff,actuatorOff)]

(sensorOn,actuatorOff) ==> (id,actuatorSwitch,id) = [(sensorOn,actuatorOn)]

(sensorOn,actuatorOn) ==> (id,sensorSwitch,id) = [(sensorOff,actuatorOn)]

(sensorOn,actuatorOn) ==> (id,actuatorSwitch,id) = [(sensorOn,actuatorOff)]
```

[40]:
```
print $ take 10 $ getTraces plant
```

```
[([],(False,False)),([sensorSwitch,sensorSwitch],(False,False)),([actuatorSwitch,actuatorSwitcl
```

We then apply `simpleRequirement2` by synchronizing the plant with the requirement.

[41]:
```
supervisor = syncEventSystems jointComposition plant simpleRequirement2

putStr $ show supervisor
```

```
Controllable Events: [actuatorSwitch]
Uncontrollable Events: [sensorSwitch]
Domain: [(False,False),(False,True),(True,False),(True,True)]
Initial values: [(False,False)]

Machine:
```

```
States: [(sensorOff,actuatorOff,r0),(sensorOff,actuatorOff,r1),(sensorOff,actuatorOn,r0),(sens
Initial States: [(sensorOff,actuatorOff,r0)]
Terminal States: [(sensorOff,actuatorOff,r0)]
Alphabet: [
    (id,actuatorSwitch,id),
    (id,sensorSwitch,id),
    (id,sensorSwitch,id||i1),
    (id,sensorSwitch,id||i2),
]

Transitions:
(sensorOff,actuatorOff,r0) ==> (id,sensorSwitch,id||i1) = [(sensorOn,actuatorOff,r0)]

(sensorOff,actuatorOff,r0) ==> (id,sensorSwitch,id||i2) = [(sensorOn,actuatorOff,r1)]

(sensorOff,actuatorOff,r1) ==> (id,actuatorSwitch,id) = [(sensorOff,actuatorOn,r0)]

(sensorOff,actuatorOff,r1) ==> (id,sensorSwitch,id) = [(sensorOn,actuatorOff,r1)]

(sensorOff,actuatorOn,r0) ==> (id,sensorSwitch,id||i1) = [(sensorOn,actuatorOn,r0)]

(sensorOff,actuatorOn,r0) ==> (id,sensorSwitch,id||i2) = [(sensorOn,actuatorOn,r1)]

(sensorOff,actuatorOn,r1) ==> (id,actuatorSwitch,id) = [(sensorOff,actuatorOff,r0)]

(sensorOff,actuatorOn,r1) ==> (id,sensorSwitch,id) = [(sensorOn,actuatorOn,r1)]

(sensorOn,actuatorOff,r0) ==> (id,sensorSwitch,id||i1) = [(sensorOff,actuatorOff,r0)]

(sensorOn,actuatorOff,r0) ==> (id,sensorSwitch,id||i2) = [(sensorOff,actuatorOff,r1)]

(sensorOn,actuatorOff,r1) ==> (id,actuatorSwitch,id) = [(sensorOn,actuatorOn,r0)]

(sensorOn,actuatorOff,r1) ==> (id,sensorSwitch,id) = [(sensorOff,actuatorOff,r1)]

(sensorOn,actuatorOn,r0) ==> (id,sensorSwitch,id||i1) = [(sensorOff,actuatorOn,r0)]

(sensorOn,actuatorOn,r0) ==> (id,sensorSwitch,id||i2) = [(sensorOff,actuatorOn,r1)]

(sensorOn,actuatorOn,r1) ==> (id,actuatorSwitch,id) = [(sensorOn,actuatorOff,r0)]

(sensorOn,actuatorOn,r1) ==> (id,sensorSwitch,id) = [(sensorOff,actuatorOn,r1)]
```

[42]:
```
print $ take 10 $ getTraces supervisor
```

```
[([],(False,False)),([sensorSwitch,actuatorSwitch,sensorSwitch,sensorSwitch,actuatorSwitch,sens
```

3.2 Restrictions

We now define a type for *Restriction* as discussed in Chapter 6, which are tuples with an event as the first element and a state as the second element.

[43]:
```haskell
type Restriction a = (BinaryRel a, StateLabel)
```

The restriction operator (↓) can then be defined as follows.

[44]:
```haskell
(\/) :: System d -> [Restriction d] -> System d
(\/)
    (System (Automaton qs is ts tp delta) contrEvents uncontrEvents dom initData)
    restrictions
  =
    System (Automaton qs is ts tp delta') contrEvents uncontrEvents dom initData
  where
    delta' q t@(r1, e, r2)
        | (e, q) `elem` restrictions = []
        | otherwise                  = delta q t
```

As an example, we will apply a restriction to the earlier defined plant.

[45]:
```haskell
restrictedPlant = plant \/ [(sensorSwitchEvent', JointLabel (SingleLabel "sensorOff") (SingleLabel "actuatorOff")), (sensorSwitchEvent', JointLabel (SingleLabel "sensorOn") (SingleLabel "actuatorOn"))]
    where
        sensorSwitchEvent' = projectLeft disjointComposition sensorSwitchEvent

print restrictedPlant
```

```
Controllable Events: [actuatorSwitch]
Uncontrollable Events: [sensorSwitch]
Domain: [(False,False),(False,True),(True,False),(True,True)]
Initial values: [(False,False)]

Machine:
States: [(sensorOff,actuatorOff),(sensorOff,actuatorOn),(sensorOn,actuatorOff),(sensorOn,actua
Initial States: [(sensorOff,actuatorOff)]
Terminal States: [(sensorOff,actuatorOff)]
Alphabet: [
    (id,sensorSwitch,id),
    (id,actuatorSwitch,id),
]

Transitions:
(sensorOff,actuatorOff) ==> (id,actuatorSwitch,id) = [(sensorOff,actuatorOn)]

(sensorOff,actuatorOn) ==> (id,sensorSwitch,id) = [(sensorOn,actuatorOn)]
```

```
(sensorOff,actuatorOn) ==> (id,actuatorSwitch,id) = [(sensorOff,actuatorOff)]

(sensorOn,actuatorOff) ==> (id,sensorSwitch,id) = [(sensorOff,actuatorOff)]

(sensorOn,actuatorOff) ==> (id,actuatorSwitch,id) = [(sensorOn,actuatorOn)]

(sensorOn,actuatorOn) ==> (id,actuatorSwitch,id) = [(sensorOn,actuatorOff)]
```

3.3 Supervisory Control

In this section we will discuss the supervisory synthesis algorithm for *Systems* as defined in chapter 8. We implement the following.

- The reduction procedure, which reduces an *System* to the update sets C and U and the finalization predicate F.
- The supervisory control algorithm in a functional style, which synthesizes the update set C' given C and U and F.
- Construction of the supervisor given C', U and F.

We first implement the reduction procedure `reduceSystem`, which reduces an *System* to the update sets C and U, and the finalization predicate F.

[46]:
```
reduceSystem
  :: Eq d
  => System d
  -> ( [EventUpdate (d, StateLabel)]
     , [EventUpdate (d, StateLabel)]
     , (d, StateLabel) -> Bool
     )
reduceSystem
    (System (Automaton qs is ts tp (==>)) controllables uncontrollables dom␣
 ↪initialValues)
  =
    (c, u, finalization)
  where
    pl' = projectLeft disjointComposition
    c = do t@(r1, e, r2) <- tp
           guard (e `elem` controllables)
           let stateRel = BinaryRel "stateRel" (\(v,q) -> map (v,) (q ==> t))
           return (pl' r1, pl' e, pl' r2 <> stateRel)
    u = do t@(r1, e, r2) <- tp
           guard (e `elem` uncontrollables)
           let stateRel = BinaryRel "stateRel" (\(v,q) -> map (v,) (q ==> t))
           return (pl' r1, pl' e, pl' r2 <> stateRel)
    finalization (v, q) = q `elem` ts
```

For the supervisory synthesis algorithm, we first define the function `leastFixpoint`, which will be used to iterate an endofunction until the result does not change anymore (according to a given function for testing on equivalence).

[47]:
```
leastFixpoint :: (a -> a -> Bool) -> (a -> a) -> a -> a
leastFixpoint eq f = rec
    where
        rec x | eq x (f x) = x
              | otherwise  = rec (f x)
```

Using `leastFixpoint` we can implement the function `fixPredicate` as shown in the algorithm.

[48]:
```
fixPredicate :: [d] -> [EventUpdate d] -> (d -> Bool) -> (d -> Bool)
fixPredicate vals binaryRels = leastFixpoint eq updatePredicate
    where
        eq p1 p2 = all (\v -> p1 v == p2 v) vals
        updatePredicate predicate' = \v -> predicate' v ||
                                            any (\r -> any predicate' 
 ↪(getEventUpdateRel r v)) binaryRels
```

```
Line 5: Redundant lambda
Found:
updatePredicate predicate'
  = \ v ->
      predicate' v ||
        any (\ r -> any predicate' (getEventUpdateRel r v)) binaryRels
Why not:
updatePredicate predicate' v
  = predicate' v ||
      any (\ r -> any predicate' (getEventUpdateRel r v)) binaryRels
```

`updateGuards` implements the contents of the while-loop of the algorithm, which updates the guard predicates for each iteration. Note that the blocking predicates for the iteration are also returned, since we need B_{j-1} in the inner while loop of the algorithm.

[49]:
```
updateGuards
  :: Show d
  => [d]
  -> [EventUpdate d]
  -> (d -> Bool)
  -> ( [EventUpdate d]
     , d -> Bool
     )
  -> ( [EventUpdate d]
     , d -> Bool
     )
updateGuards
    vals
```

```
    uncontrollables
    finalization
    (controllables, badPredicate)
  =
    (controllables', badPredicate')
  where
    nonBlocking = fixPredicate vals (uncontrollables ++ controllables)␣
↪finalization
    initialBad v = badPredicate v || not (nonBlocking v)
    badPredicate' = fixPredicate vals uncontrollables initialBad
    guardLabel = intercalate " && " $ map (("not "++).show) $ filter␣
↪badPredicate' vals
    updateGuard (rel1, e, rel2) = (rel1, e, rel2 <> BinaryRel guardLabel (\v ->␣
↪[v | not (badPredicate' v)]))
    controllables' = map updateGuard controllables
```

Lastly, `synthesizeSupervisorUpdates` will compute the update set C' given C, U and F.

[50]:
```
synthesizeSupervisorUpdates
    :: ( Eq d
       , Show d
       )
    => [d]
    -> [EventUpdate d]
    -> [EventUpdate d]
    -> (d -> Bool)
    -> [EventUpdate d]
synthesizeSupervisorUpdates
    domain
    controllables
    uncontrollables
    finalization
  =
    controllables'
  where
    initialBad d = False
    eq (contr1, bad1) (contr2, bad2)
      =
        all (\(u1, u2) -> all (\v -> getEventUpdateRel u1 v ==␣
↪getEventUpdateRel u2 v) domain) (zip contr1 contr2)
    (controllables', bad)
      =
        leastFixpoint eq (updateGuards domain uncontrollables finalization)␣
↪(controllables, initialBad)
```

`constructSupervisor` constructs the supervisor from, among other things, the synthesized update set C'.

[51]:
```
constructSupervisor
    :: Eq d
    => [EventUpdate (d, StateLabel)]
    -> [EventUpdate (d, StateLabel)]
    -> [BinaryRel d]
    -> [BinaryRel d]
    -> [(d, StateLabel)]
    -> [d]
    -> [StateLabel]
    -> System (d, StateLabel)
constructSupervisor
    c
    u
    contrEvents
    uncontrEvents
    dom
    initVals
    initStates
  =
    System machine
           contrEvents'
           uncontrEvents'
           dom
           initials'
  where
    q = SingleLabel "q"
    pl' = projectLeft disjointComposition
    contrEvents' = [pl' e | e <- contrEvents]
    uncontrEvents' = [pl' e | e <- uncontrEvents]
    machine = Automaton [q] [q] [q] tp delta
    tp = c ++ u
    delta q t = [q]
    initials' = [(v, q') | v <- initVals, q' <- initStates]
```

Lastly, we will combine all discussed elements into the function `synthesizeSupervisor`, which implements the complete supervisory synthesis algorithm for *EventSystems*.

[52]:
```
synthesizeSupervisor :: (Eq d, Show d) => System d -> System (d, StateLabel)
synthesizeSupervisor
    es@(System machine controllables uncontrollables dom initialValues)
  =
    supervisor
  where
    (Automaton qs is ts tp (==>)) = machine
    domain' = [(v, q) | v <- dom, q <- qs]
    (c, u, finalization) = reduceSystem es
    c' = synthesizeSupervisorUpdates domain' c u finalization
```

```
    supervisor = constructSupervisor c' u controllables uncontrollables domain'␣
↪initialValues is
```

supervise computes the supervisor for a given system, and synchronizes the resulting supervisor with said system.

[53]:
```
supervise :: (Eq d, Show d) => System d -> System (d, StateLabel)
supervise system = syncEventSystems domComp system supervisor
  where
    supervisor = synthesizeSupervisor system
    domComp = DomainComposition
        { combine = \v1 v2 -> v2
        , decompose = \(d, q) -> (d, (d, q))
        , checkComp = \d (d', q) -> d == d'
        , extract1 = fst
        , extract2 = id
        , augment1 = \(d, q) d' -> (d', q)
        , augment2 = \t t' -> t'
        }
```

3.3.1 Examples

This example is based on Example 6.9.1.

[54]:
```
supervisorExample1 :: System (Bool, Bool, Int)
supervisorExample1 = System m contr uncontr dom initialData
    where
        dom = [(bs,ba,n) | bs <- [True, False], ba <- [True, False], n <- [0..
↪10]]

        s_switch = BinaryRel "s_switch" (\(bs,ba,n) -> [(not bs, ba ,n)])
        a_switch = BinaryRel "a_switch" (\(bs,ba,n) -> [(bs, not ba ,n)])
        contr = [a_switch]
        uncontr = [s_switch]
        initialData = [(False,False,0)]

        g1 = BinaryRel "g1" (\(bs,ba,n) -> [(bs,ba,n) | n < 8])
        g2 = BinaryRel "g2" (\(bs,ba,n) -> [(bs,ba,n) | n < 9])
        g3 = BinaryRel "g3" (\(bs,ba,n) -> [(bs,ba,n) | n < 7])
        u = BinaryRel "u" (\(bs, ba,n) -> [(bs, ba, n + 1)])

        m = Automaton [q0, q1, q2] [q0] [q0] tp delta
        [q0, q1, q2] = map SingleLabel ["q0", "q1", "q2"]
        tp = [(g1, a_switch, u), (g2, a_switch, u), (g3, s_switch, u)]
        delta q t = show q ==> show t

        "q0" ==> "(g1,a_switch,u)" = [q1]
```

```
        "q1" ==> "(g2,a_switch,u)" = [q2]
        "q2" ==> "(g3,s_switch,u)" = [q0]
        _ ==> _ = []
```

[55]:
```
supervisedSystem = supervise supervisorExample1
print supervisedSystem
```

```
Controllable Events: [a_switch]
Uncontrollable Events: [s_switch]
Domain: [((True,True,0),q0),((True,True,0),q1),((True,True,0),q2),((True,True,1),q0),((True,Tr
Initial values: [((False,False,0),q0)]

Machine:
States: [(q0,q),(q1,q),(q2,q)]
Initial States: [(q0,q)]
Terminal States: [(q0,q)]
Alphabet: [
    (g1,a_switch,u||u;stateRel;not ((True,Tr...,
    (g1||g2,a_switch,u||u;stateRel;not ((Tru...,
    (g2||g1,a_switch,u||u;stateRel;not ((Tru...,
    (g2,a_switch,u||u;stateRel;not ((True,Tr...,
    (g3,s_switch,u||u;stateRel),
]

Transitions:
(q0,q) ==> (g1,a_switch,u||u;stateRel;not ((True,Tr... = [(q1,q)]

(q0,q) ==> (g1||g2,a_switch,u||u;stateRel;not ((Tru... = [(q1,q)]

(q1,q) ==> (g2||g1,a_switch,u||u;stateRel;not ((Tru... = [(q2,q)]

(q1,q) ==> (g2,a_switch,u||u;stateRel;not ((True,Tr... = [(q2,q)]

(q2,q) ==> (g3,s_switch,u||u;stateRel) = [(q0,q)]
```

This example is based on Example 6.9.2.

[56]:
```
supervisorExample2 :: System Int
supervisorExample2 = System m contr uncontr dom initialData
    where
        dom = [0..5]

        cTakeOne = BinaryRel "cTakeOne" (\n -> [n - 1])
        cTakeTwo = BinaryRel "cTakeTwo" (\n -> [n - 2])
        uTakeOne = BinaryRel "uTakeOne" (\n -> [n - 1])
        uTakeTwo = BinaryRel "uTakeTwo" (\n -> [n - 2])
```

```
        contr = [cTakeOne, cTakeTwo]
        uncontr = [uTakeOne, uTakeTwo]
        initialData = [3]

        g1 = BinaryRel "g1" (\n -> [n | n > 1])
        g2 = BinaryRel "g2" (\n -> [n | n > 2])
        g3 = BinaryRel "g3" (\n -> [n | n == 1])
        g4 = BinaryRel "g4" (\n -> [n | n == 2])

        m = Automaton [q0, q1, q2, q3] [q0] [q3] tp delta
        [q0, q1, q2, q3] = map SingleLabel ["q0", "q1", "q2", "q3"]
        tp = [(g1, cTakeOne,identityRel), (g2, cTakeTwo, identityRel),
              (g1, uTakeOne, identityRel), (g2, uTakeTwo, identityRel),
              (g4, cTakeTwo, identityRel), (g3, cTakeOne, identityRel),
              (g4, uTakeTwo, identityRel), (g3, uTakeOne, identityRel)]
        delta q t = show q ==> show t

        "q0" ==> "(g1,cTakeOne,id)" = [q1]
        "q0" ==> "(g2,cTakeTwo,id)" = [q1]
        "q1" ==> "(g1,uTakeOne,id)" = [q0]
        "q1" ==> "(g2,uTakeTwo,id)" = [q0]
        "q0" ==> "(g3,cTakeOne,id)" = [q2]
        "q0" ==> "(g4,cTakeTwo,id)" = [q2]
        "q1" ==> "(g3,uTakeOne,id)" = [q3]
        "q1" ==> "(g4,uTakeTwo,id)" = [q3]
        _ ==> _ = []
```

[57]:
```
supervisedSystem2 = supervise supervisorExample2
print supervisedSystem2
```

```
Controllable Events: [cTakeOne,cTakeTwo]
Uncontrollable Events: [uTakeOne,uTakeTwo]
Domain: [(0,q0),(0,q1),(0,q2),(0,q3),(1,q0),(1,q1),(1,q2),(1,q3),(2,q0),(2,q1),(2,q2),(2,q3),(
Initial values: [(3,q0)]

Machine:
States: [(q0,q),(q1,q),(q2,q),(q3,q)]
Initial States: [(q0,q)]
Terminal States: [(q3,q)]
Alphabet: [
    (g1,cTakeOne,id||id;stateRel;not (0,q0) ...,
    (g1||g3,cTakeOne,id||id;stateRel;not (0,...,
    (g3||g1,cTakeOne,id||id;stateRel;not (0,...,
    (g3,cTakeOne,id||id;stateRel;not (0,q0) ...,
    (g2,cTakeTwo,id||id;stateRel;not (0,q0) ...,
    (g2||g4,cTakeTwo,id||id;stateRel;not (0,...,
    (g4||g2,cTakeTwo,id||id;stateRel;not (0,...,
```

```
    (g4,cTakeTwo,id||id;stateRel;not (0,q0) ...,
    (g1,uTakeOne,id||id;stateRel),
    (g1||g3,uTakeOne,id||id;stateRel),
    (g3||g1,uTakeOne,id||id;stateRel),
    (g3,uTakeOne,id||id;stateRel),
    (g2,uTakeTwo,id||id;stateRel),
    (g2||g4,uTakeTwo,id||id;stateRel),
    (g4||g2,uTakeTwo,id||id;stateRel),
    (g4,uTakeTwo,id||id;stateRel),
]

Transitions:
(q0,q) ==> (g1,cTakeOne,id||id;stateRel;not (0,q0) ... = [(q1,q)]

(q0,q) ==> (g1||g3,cTakeOne,id||id;stateRel;not (0,... = [(q1,q)]

(q0,q) ==> (g3||g1,cTakeOne,id||id;stateRel;not (0,... = [(q2,q)]

(q0,q) ==> (g3,cTakeOne,id||id;stateRel;not (0,q0) ... = [(q2,q)]

(q0,q) ==> (g2,cTakeTwo,id||id;stateRel;not (0,q0) ... = [(q1,q)]

(q0,q) ==> (g2||g4,cTakeTwo,id||id;stateRel;not (0,... = [(q1,q)]

(q0,q) ==> (g4||g2,cTakeTwo,id||id;stateRel;not (0,... = [(q2,q)]

(q0,q) ==> (g4,cTakeTwo,id||id;stateRel;not (0,q0) ... = [(q2,q)]

(q1,q) ==> (g1,uTakeOne,id||id;stateRel) = [(q0,q)]

(q1,q) ==> (g1||g3,uTakeOne,id||id;stateRel) = [(q0,q)]

(q1,q) ==> (g3||g1,uTakeOne,id||id;stateRel) = [(q3,q)]

(q1,q) ==> (g3,uTakeOne,id||id;stateRel) = [(q3,q)]

(q1,q) ==> (g2,uTakeTwo,id||id;stateRel) = [(q0,q)]

(q1,q) ==> (g2||g4,uTakeTwo,id||id;stateRel) = [(q0,q)]

(q1,q) ==> (g4||g2,uTakeTwo,id||id;stateRel) = [(q3,q)]

(q1,q) ==> (g4,uTakeTwo,id||id;stateRel) = [(q3,q)]
```

Appendix D

Syntax Implementation

In this appendix the implementation of the syntax of X-Control is discussed in more detail.

[1]:
```
:load SemanticDomain.hs
```

[2]:
```
import qualified Data.Map.Strict as Map hiding (foldl, filter, take)
import Control.Monad.State
import Data.List hiding (insert, union)
import Data.Maybe
import qualified Data.Set

import SemanticDomain
```

1 Automata

We start with the introduction of syntax used to define an automaton.

We first define the declarations that can be made for an automaton. For this we define the datatype `AutomatonDeclarationStatement a`, which will be used to define an automaton with type `a`. We define the following declaration statements:

- A statement for declaring state which is not initial nor terminal: `State "stateName"`,
- a statement for declaring an initial state which is not terminal: `InitialState "stateName"`,
- a statement for declaring a terminal state which is not initial state: `TerminalState "stateName"`,
- a statement for declaring a state which is both initial and terminal: `InitialTerminalState "stateName"`,
- and a statement for declaring a edge from one state to another, with symbol `x` of type `a`: `Edge from "originState" to "targetState" with x`.

We also define some keywords which will be given as arguments to the `Edge` constructor.

[3]:
```
data AutomatonDeclarationStatement a = State String | InitialState String |␣
↪TerminalState String
                                     | InitialTerminalState String | Edge␣
↪KeyWord String KeyWord String KeyWord a

type KeyWord = String

from :: KeyWord
from = "from"
```

```haskell
to :: KeyWord
to = "to"

with :: KeyWord
with = "with"
```

A definition for an automaton is then a list of automaton declarations.

[4]:
```haskell
type AutomatonDeclaration a = [AutomatonDeclarationStatement a]
```

The function `automaton` is then a valuation function for automata, which constructs an automaton (as defined in the semantic domain), from an automaton declaration.

[5]:
```haskell
automaton :: (AutomatonType a, Eq a) => AutomatonDeclaration a  -> Automaton a␣
 ↪StateLabel
automaton automatonDeclarations = Automaton qs is ts alph delta
  where
    qs = map f $ filter g automatonDeclarations
      where
        g (State name) = True
        g (InitialState name) = True
        g (TerminalState name) = True
        g (InitialTerminalState name) = True
        g _ = False

        f (State name) = SingleLabel name
        f (InitialState name) = SingleLabel name
        f (TerminalState name) = SingleLabel name
        f (InitialTerminalState name) = SingleLabel name


    is = map f $ filter g automatonDeclarations
      where
        g (InitialState name) = True
        g (InitialTerminalState name) = True
        g _ = False

        f (InitialState name) = SingleLabel name
        f (InitialTerminalState name) = SingleLabel name

    ts = map f $ filter g automatonDeclarations
      where
        g (TerminalState name) = True
        g (InitialTerminalState name) = True
        g _ = False

        f (TerminalState name) = SingleLabel name
```

```
        f (InitialTerminalState name) = SingleLabel name

    alph = Data.Set.toList . Data.Set.fromAscList $ map f $ filter g␣
↪automatonDeclarations
      where
        g (Edge "from" origin "to" target "with" label) = True
        g _ = False

        f (Edge "from" origin "to" target "with" label) = label

    delta state symbol = map f $ filter g automatonDeclarations
      where
        g (Edge "from" origin "to" target "with" label) = state == SingleLabel␣
↪origin && label == symbol
        g _ = False

        f (Edge "from" origin "to" target "with" label) = SingleLabel target
```

As an example we define an automaton which behavior contains all sequences belonging to the regular expression $(ab)^*c$.

[6]:
```
exampleAutomaton :: Automaton Char StateLabel
exampleAutomaton = automaton [
    InitialState "q0",
    State "q1",
    TerminalState "q2",

    Edge from "q0" to "q1" with 'a',
    Edge from "q1" to "q0" with 'b',
    Edge from "q0" to "q2" with 'c'
    ]
```

[7]:
```
take 10 $ getLanguage exampleAutomaton
```

```
[(q2,"c"),(q2,"abc"),(q2,"ababc"),(q2,"abababc"),(q2,"ababababc"),(q2,"abababababc"),(q2,"ababa
```

2 Relations

Next we will define syntax to define the binary relations as used in X-machines and D-systems.

We first define syntax for defining an element of our domain. For now, we will only support Boolean and integer elements. A Boolean element is defined as `BoolElement "elementId" initialValue`, and a integer element is defined as `IntElement "elementId" intRange initialValue`. An `ElementId` is the identifier for the domain element.

[8]:
```
type ElementId = String

data DomainElement = BoolElement ElementId Bool | IntElement ElementId [Int] Int

getElementId :: DomainElement -> ElementId
getElementId (BoolElement elementId value) = elementId
getElementId (IntElement elementId domain value) = elementId
```

The datatype `DomainElementValue` represents a current value in the domain.

[9]:
```
data DomainElementValue = BoolValue Bool | IntValue Int deriving Eq

instance Show DomainElementValue
  where
    show (BoolValue b) = show b
    show (IntValue x) = show x
```

`declareElement` is a valuation function for domain elements.

[10]:
```
declareElement :: DomainElement -> DomainElementValue
declareElement (BoolElement elementId initialValue) = BoolValue initialValue
declareElement (IntElement elementId domain initialValue) = IntValue␣
 ↪initialValue
```

`getDomainElementDomain` computes the domain for a given `DomainElement`, which is the list of possible values which the element can have.

[11]:
```
getDomainElementDomain :: DomainElement -> [DomainElementValue]
getDomainElementDomain (BoolElement elementId initialValue) = [BoolValue True,␣
 ↪BoolValue False]
getDomainElementDomain (IntElement elementId domain initialValue) = map␣
 ↪IntValue domain
```

We define functions to obtain the `Int` and `Bool` values from `DomainElementValue`.

[12]:
```
getIntFromElementValue :: DomainElementValue -> Int
getIntFromElementValue (IntValue x) = x
getIntFromElementValue (BoolValue b) = error "not an int value"

getBoolFromElementValue :: DomainElementValue -> Bool
getBoolFromElementValue (BoolValue b) = b
getBoolFromElementValue (IntValue x) = error "not a bool value"
```

We also define functions to change the values of `DomainElementValue` instances.

[13]:
```
setIntToElementValue :: DomainElementValue -> Int -> DomainElementValue
setIntToElementValue (IntValue x) = IntValue
setIntToElementValue (BoolValue b) = error "not an int value"
```

```
setBoolToElementValue :: DomainElementValue -> Bool -> DomainElementValue
setBoolToElementValue (BoolValue b) = BoolValue
setBoolToElementValue (IntValue x) = error "not a bool value"
```

The type of our domain, modeled by `DomainValue`, is a mapping from elementIds to the corresponding `DomainElementValue`.

[14]:
```
type DomainValue = Map.Map ElementId DomainElementValue
```

With `declareDomain` we can instantiate a domain given a `DomainElement` list.

[15]:
```
declareDomain :: [DomainElement] -> DomainValue
declareDomain domainElements = Map.fromList $ map (\de -> (getElementId de,␣
↪declareElement de)) domainElements
```

`getPossibleDomainValues` computes all possible instances of a domain given as a `DomainElement` list.

[16]:
```
fullCartesianProduct :: [[a]] -> [[a]]
fullCartesianProduct [] = []
fullCartesianProduct [xs] = map (:[]) xs
fullCartesianProduct (xs:xss) = [ x:xs' | x <- xs, xs' <- fullCartesianProduct␣
↪xss]

getPossibleDomainValues :: [DomainElement] -> [DomainValue]
getPossibleDomainValues [] = [Map.fromList []]
getPossibleDomainValues domainElements = map Map.fromList $␣
↪fullCartesianProduct elementDomains
  where
    elementDomains = [ map (getElementId de,) $ getDomainElementDomain de
                     | de <- domainElements
                     ]
```

We define a functions for obtaining values from a `DomainValue`, given an `ElementId`.

[17]:
```
getIntFromDomainValue :: DomainValue -> ElementId -> Int
getIntFromDomainValue domValue elementId = getIntFromElementValue $ domValue␣
↪Map.! elementId

getBoolFromDomainValue :: DomainValue -> ElementId -> Bool
getBoolFromDomainValue domValue elementId = getBoolFromElementValue $ domValue␣
↪Map.! elementId
```

And we define functions for changing a value in a `DomainValue` belonging to some `ElementId`.

[18]:
```
setIntInDomainValue :: DomainValue -> ElementId -> Int -> DomainValue
setIntInDomainValue domValue elementId x = Map.adjust (`setIntToElementValue`␣
↪x) elementId domValue
```

```
setBoolInDomainValue :: DomainValue -> ElementId -> Bool -> DomainValue
setBoolInDomainValue domValue elementId b = Map.adjust (`setBoolToElementValue`␣
↪b) elementId domValue
```

For D-Systems we need syntax for defining relations which updates the domain value. For this we define the type `DomainState` which is based on the type `State DomainValue`. We use the `State a` monad since it facilitates a method to manipulate the state domain in a procedural way. The do-notation in Haskell essentially provides us most of the syntax for defining relations.

[19]:
```
newtype DomainState a = DomainState { getState :: State DomainValue a }

instance Functor DomainState
  where
    fmap f (DomainState s) = DomainState $ fmap f s

instance Applicative DomainState
  where
    pure x = DomainState (pure x)
    (DomainState fSt) <*> (DomainState xSt) = DomainState $ fSt <*> xSt

instance Monad DomainState
  where
    return x = DomainState (return x)
    (DomainState s1) >>= f = DomainState $ s1 >>= (getState . f)
```

We define functions for retrieving values from domain elements within the `DomainState` environment. These functions are part of our syntax.

[20]:
```
getIntValue :: ElementId -> DomainState Int
getIntValue elementId = DomainState $ get >>= (\domainValue ->
    return $ getIntFromDomainValue domainValue elementId)

getBoolValue :: ElementId -> DomainState Bool
getBoolValue elementId = DomainState $ get >>= (\domainValue ->
    return $ getBoolFromDomainValue domainValue elementId)
```

We also define functions for setting values to domain elements withing the `DomainState` environment. These functions are also part of our syntax.

[21]:
```
setIntValue :: ElementId -> Int -> DomainState ()
setIntValue elementId value = DomainState $ get >>= (\domainValue ->
    let newDomainValue = setIntInDomainValue domainValue elementId value
    in put newDomainValue)

setBoolValue :: ElementId -> Bool -> DomainState ()
setBoolValue elementId value = DomainState $ get >>= (\domainValue ->
    let newDomainValue = setBoolInDomainValue domainValue elementId value
```

```
    in put newDomainValue)
```

`function` is a valuation function for `DomainState` without a result. We now have syntax for creating relations with exactly 1 result for some domain value.

[22]:
```
function :: String -> DomainState () -> BinaryRel DomainValue
function name domainState = BinaryRel name (\d -> [(execState . getState) domainState d])
```

What follows is an example function.

[23]:
```
exampleFunction = function "exampleFunction" $ do {
    testInt <- getIntValue "testInt";
    testBool <- getBoolValue "testBool";

    if testBool then
        setIntValue "testInt" (testInt + 2);
    else
        setIntValue "testInt" (testInt + 3);
    }
```

`guard` is a valuation function for `DomainState` with a Boolean result. This gives us syntax for creating relations with, for some domain value, the same domain value as result, or no result. These relations essentially act as (transition) guards.

[24]:
```
guard :: String -> DomainState Bool -> BinaryRel DomainValue
guard name domainState = BinaryRel name (\d -> [d | (evalState . getState) domainState d])
```

What follows is an example guard.

[25]:
```
exampleGuard = guard "exampleGuard" $ do {
    testInt <- getIntValue "testInt";

    return $ testInt < 5;
    }
```

`relation` is a valuation function for `DomainState [DomainState ()]`. The function gives us syntax for defining relations which can have multiple results for some domain values. Note that functions and guards can also be implemented using `relation`.

[26]:
```
relation :: String -> DomainState [DomainState ()] -> BinaryRel DomainValue
relation name domainState = BinaryRel name r
  where
    r domainValue = map (\ds -> (execState . getState) (domainState >> ds) domainValue) domainStateOptions
      where
        domainStateOptions = (evalState . getState) domainState domainValue
```

What follows is an example relation.

[27]:
```
exampleRelation = relation "exampleRelation" $ do {
    testBool <- getBoolValue "testBool";

    if testBool then
        return $ map (setIntValue "testInt") [1..3];
    else
        return $ map (setIntValue "testInt") [3..5];
    }
```

3 Systems

We will now create syntax for D-Systems, in which we use the previously discussed syntax for automata and relations.

The datatype `RelationDeclaration` will be used as syntax to define relations belonging to some D-System.

[28]:
```
data RelationDeclaration = Function String (DomainState ())
                         | Guard String (DomainState Bool)
                         | Relation String (DomainState [DomainState ()])
```

`declareRel` constructs a binary relation from a `RelationDeclaration`, using the appropriate valuation function.

[29]:
```
declareRel :: RelationDeclaration -> BinaryRel DomainValue
declareRel (Function name domainState) = function name domainState
declareRel (Guard name domainState) = guard name domainState
declareRel (Relation name domainState) = relation name domainState
```

The datatype `SystemSpecification` will be used as syntax for specifying D-Systems. The field-labels notation from Haskell provides us with syntax to specify the different elements of a D-System. To define the machine of the system, we use the syntax for automaton with type `(String, String, String)`.

[30]:
```
data SystemSpecification = SystemSpecification
    { domainElements :: [DomainElement]
    , controllableEvents :: [RelationDeclaration]
    , uncontrollableEvents :: [RelationDeclaration]
    , otherOperations :: [RelationDeclaration]
    , machine :: AutomatonDeclaration (String, String, String)
    }
```

`declareSystem` is the valuation function for `SystemSpecification`.

[31]:
```
declareSystem :: SystemSpecification -> System DomainValue
declareSystem systemSpecification = System mach contrs uncontrs dom initialVals
  where
```

```
    dom = getPossibleDomainValues $ domainElements systemSpecification
    initialVals = [declareDomain $ domainElements systemSpecification]
    contrs = map declareRel $ controllableEvents systemSpecification
    uncontrs = map declareRel $ uncontrollableEvents systemSpecification
    otherOps = map declareRel $ otherOperations systemSpecification

    relationMap = Map.insert "id" mempty . Map.fromList $ map (\br@(BinaryRel␣
↪name rel) -> (name, br)) $ contrs ++ uncontrs ++ otherOps
    getRel name | name `Map.member` relationMap = relationMap Map.! name
                | otherwise                     = error $ "relation " ++ name␣
↪++ " unknown" ++ (show contrs) ++ (show uncontrs)
    machineDecl = map f $ machine systemSpecification
      where
        f (Edge "from" origin "to" target "with" (r1name, ename, r2name))
          =
            Edge from origin to target with (getRel r1name, getRel ename,␣
↪getRel r2name)
        f (State name) = State name
        f (InitialState name) = InitialState name
        f (TerminalState name) = TerminalState name
        f (InitialTerminalState name) = InitialTerminalState name
    mach = automaton machineDecl
```

```
Line 12: Redundant bracket
Found:
(show contrs) ++ (show uncontrs)
Why not:
show contrs ++ (show uncontrs)Line 12: Redundant bracket
Found:
(show contrs) ++ (show uncontrs)
Why not:
(show contrs) ++ show uncontrs
```

We now define an example system.

[32]:
```
systemExample :: System DomainValue
systemExample = declareSystem SystemSpecification
    { domainElements = [
          IntElement "coins" [0..5] 5
      ],

      controllableEvents = [
          Relation "player1Take" $ do {
              coins <- getIntValue "coins";

              return [
                  setIntValue "coins" (coins - 1),
```

```
                setIntValue "coins" (coins - 2)
            ];
            }
    ],

    uncontrollableEvents = [
        Relation "player2Take" $ do {
            coins <- getIntValue "coins";

            return [
                setIntValue "coins" (coins - 1),
                setIntValue "coins" (coins - 2)
            ];
            }
    ],

    otherOperations = [
        Guard "notLost" $ do {coins <- getIntValue "coins"; return $ coins >␣
↪0},
        Guard "lost" $ do {coins <- getIntValue "coins"; return $ coins == 0}
    ],

    machine = [
        InitialState "player1Turn",
        State "player2Turn",
        State "player1Lost",
        TerminalState "player2Lost",

        Edge from "player1Turn" to "player2Turn" with ("id", "player1Take",␣
↪"notLost"),
        Edge from "player1Turn" to "player1Lost" with ("id", "player1Take",␣
↪"lost"),
        Edge from "player2Turn" to "player1Turn" with ("id", "player2Take",␣
↪"notLost"),
        Edge from "player2Turn" to "player2Lost" with ("id", "player2Take",␣
↪"lost")
    ]
  }
```

[33]:
```
print systemExample
```

```
Controllable Events: [player1Take]
Uncontrollable Events: [player2Take]
Domain: [fromList [("coins",0)],fromList [("coins",1)],fromList [("coins",2)],fromList [("coin
Initial values: [fromList [("coins",5)]]

Machine:
```

```
States: [player1Turn,player2Turn,player1Lost,player2Lost]
Initial States: [player1Turn]
Terminal States: [player2Lost]
Alphabet: [
    (id,player1Take,notLost),
    (id,player1Take,lost),
    (id,player2Take,notLost),
    (id,player2Take,lost),
]

Transitions:
player1Turn ==> (id,player1Take,notLost) = [player2Turn]

player1Turn ==> (id,player1Take,lost) = [player1Lost]

player2Turn ==> (id,player2Take,notLost) = [player1Turn]

player2Turn ==> (id,player2Take,lost) = [player2Lost]
```

[34]:
```
take 10 $ getTraces systemExample
```

```
[([player1Take,player2Take,player1Take,player2Take],fromList [("coins",0)]),([player1Take,playe
```

4 Modules

We will introduce syntax for the concept of *Modules*. A module consists of a set of system specifications and restrictions. A module can then be reduced to a single system using the synchronous product operator as implemented in the semantic domain.

We first define syntax for declarations which can be made within in a module. As of writing, we will only support declaring systems and restrictions within a module.

[35]:
```
data ModuleDeclarationStatement = DeclareSystem String SystemSpecification
                                | DeclareRestriction String KeyWord String

restricts :: KeyWord
restricts = "restricts"
```

A module specification is then a list of `ModuleDeclarationStatement`, which forms the syntax for defining modules.

[36]:
```
type ModuleSpecification = [ModuleDeclarationStatement]
```

In order to synchronize multiple systems, some bookkeeping has to be done to refer to variables of other systems. We will introduce functions for adding and removing prefixes to and from elementIds in a `DomainValue`.

[37]:
```
addPrefixToValues :: String -> [String] -> DomainValue -> DomainValue
addPrefixToValues prefix elementNames = Map.mapKeys f
  where
    f key | key `elem` elementNames = prefix ++ "." ++ key
          | otherwise = key

removePrefixFromValues :: String -> DomainValue -> DomainValue
removePrefixFromValues prefix = Map.mapKeys (\key -> fromMaybe key (stripPrefix␣
 ↪(prefix ++ ".") key))
```

transformRelationDeclaration transforms a RelationDeclaration, such that it can be used in the context of another system.

[38]:
```
transformRelationDeclaration :: String -> [String] -> RelationDeclaration ->␣
 ↪RelationDeclaration
transformRelationDeclaration prefix elementNames (Function name domainState) =␣
 ↪Function (prefix ++ "." ++ name) domainState'
  where
    domainState' = DomainState $ do domainValue <- get
                                    put (removePrefixFromValues prefix␣
 ↪domainValue)
                                    getState domainState
                                    domainValue' <- get
                                    put (addPrefixToValues prefix elementNames␣
 ↪domainValue')

transformRelationDeclaration prefix elementNames (Guard name domainState) =␣
 ↪Guard (prefix ++ "." ++ name) domainState'
  where
    domainState' = DomainState $ do domainValue <- get
                                    put (removePrefixFromValues prefix␣
 ↪domainValue)
                                    getState domainState

transformRelationDeclaration prefix elementNames (Relation name domainState) =␣
 ↪Relation (prefix ++ "." ++ name) domainState'
  where
    domainState' = DomainState $ do domainValue <- get
                                    put (removePrefixFromValues prefix␣
 ↪domainValue)
                                    result <- getState domainState
                                    return $ map f result
      where
        f (DomainState state) = DomainState $ do state
                                                 domainValue <- get
                                                 put (addPrefixToValues prefix␣
 ↪elementNames domainValue)
```

```
Line 4: Reduce duplication
Found:
domainValue <- get
put (removePrefixFromValues prefix domainValue)
getState domainState

Why not:
Combine with -:12:37
```

getTransformedEvent takes a named specification and returns a map with keys of the form ("systemName", "eventName") and the corresponding transformed events as values.

[39]:
```haskell
getTransformedEvents :: (String, SystemSpecification) -> (Map.Map String␣
↪RelationDeclaration, Map.Map String RelationDeclaration)
getTransformedEvents (sName, systemSpec) = (transformEvents $␣
↪controllableEvents systemSpec
                                           , transformEvents $␣
↪uncontrollableEvents systemSpec
                                           )
  where
    events = controllableEvents systemSpec ++ uncontrollableEvents systemSpec
    elementNames = map getElementId $ domainElements systemSpec

    transformEvents evs = Map.fromList $ map f evs

    f r@(Function rName fn) = (sName ++ "." ++ rName,␣
↪transformRelationDeclaration sName elementNames r)
    f r@(Guard rName fn) = (sName ++ "." ++ rName, transformRelationDeclaration␣
↪sName elementNames r)
    f r@(Relation rName fn) = (sName ++ "." ++ rName,␣
↪transformRelationDeclaration sName elementNames r)
```

applyEventRefs takes a SystemSpecification and a mapping as retrieved with getTransformedEvent and adds the referenced events to the given system.

[40]:
```haskell
applyEventRefs :: SystemSpecification -> (Map.Map String RelationDeclaration,␣
↪Map.Map String RelationDeclaration) -> SystemSpecification
applyEventRefs systemSpec (externalControllableEvents,␣
↪externalUncontrollableEvents)
  = systemSpec { controllableEvents = controllableEvents'
               , uncontrollableEvents = uncontrollableEvents'
               }
  where
    (controllableEventNames, uncontrollableEventNames) = (map f␣
↪(controllableEvents systemSpec), map f (uncontrollableEvents systemSpec))
      where
        f r@(Function rName fn) = rName
```

```
      f r@(Guard rName fn) = rName
      f r@(Relation rName fn) = rName

  eventRefs = nub $ map f $ filter g $ machine systemSpec
    where
      g (Edge "from" origin "to" target "with" (r1name, ename, r2name))
        = ename /= "id" &&
        ename `notElem` (controllableEventNames ++ uncontrollableEventNames)
      g _ = False
      f (Edge "from" origin "to" target "with" (r1name, ename, r2name)) =␣
↪ename


  controllableEvents' = controllableEvents systemSpec ++ map f (filter g␣
↪eventRefs)
    where
      g = (`Map.member` externalControllableEvents)
      f = (externalControllableEvents Map.!)
  uncontrollableEvents' = uncontrollableEvents systemSpec ++ map f (filter g␣
↪eventRefs)
    where
      g = (`Map.member` externalUncontrollableEvents)
      f = (externalUncontrollableEvents Map.!)
```

syncTwoSystemSpecs takes two system specifications and returns the synchronized result of the two systems.

[41]:
```
syncTwoSystemSpecs :: (String, SystemSpecification) -> (String,␣
↪SystemSpecification) -> System DomainValue
syncTwoSystemSpecs (name1, systemSpec1) (name2, systemSpec2) = syncEventSystems␣
↪domainComposition system1 system2
 where
  elementNames1 = map getElementId $ domainElements systemSpec1
  elementNames2 = map getElementId $ domainElements systemSpec2

  system1 = declareSystem systemSpec1
  system2 = declareSystem systemSpec2

  domainComposition = DomainComposition combine decompose checkComp extract1␣
↪extract2 augment1 augment2
  combine d1 d2 = let prefixedD1 = addPrefixToValues name1 elementNames1 d1
                      prefixedD2 = addPrefixToValues name2 elementNames2 d2
                  in  Map.union prefixedD1 prefixedD2
  decompose dc = let unPrefixedD1 = removePrefixFromValues name1 dc
                     unPrefixedD2 = removePrefixFromValues name2 dc
                 in (unPrefixedD1, unPrefixedD2)
  checkComp d1 d2 = let prefixedD1 = addPrefixToValues name1 elementNames1 d1
```

```haskell
                          prefixedD2 = addPrefixToValues name2 elementNames2 d2
                      in prefixedD1 == prefixedD2 || (Map.size (Map.
↪intersection prefixedD1 prefixedD2) == 0)
    extract1 dc = removePrefixFromValues name1 dc
    extract2 dc = removePrefixFromValues name2 dc
    augment1 dc d1 = addPrefixToValues name1 elementNames1 d1
    augment2 dc d2 = addPrefixToValues name2 elementNames2 d2
```

```
Line 20: Eta reduce
Found:
extract1 dc = removePrefixFromValues name1 dc
Why not:
extract1 = removePrefixFromValues name1Line 21: Eta reduce
Found:
extract2 dc = removePrefixFromValues name2 dc
Why not:
extract2 = removePrefixFromValues name2Line 22: Eta reduce
Found:
augment1 dc d1 = addPrefixToValues name1 elementNames1 d1
Why not:
augment1 dc = addPrefixToValues name1 elementNames1Line 23: Eta reduce
Found:
augment2 dc d2 = addPrefixToValues name2 elementNames2 d2
Why not:
augment2 dc = addPrefixToValues name2 elementNames2
```

`syncToExistingSystem` takes a named system specification and a existing system, and synchronizes the system resulting from the specification with the given system.

[42]:
```haskell
syncToExistingSystem :: (String, SystemSpecification) -> System DomainValue ->␣
↪System DomainValue
syncToExistingSystem (sysName, systemSpec) system = syncEventSystems␣
↪domainComposition system1 system
  where
    elementNames = map getElementId $ domainElements systemSpec

    system1 = declareSystem systemSpec

    domainComposition = DomainComposition combine decompose checkComp extract1␣
↪extract2 augment1 augment2
    combine d1 d2 = let prefixedD1 = addPrefixToValues sysName elementNames d1
                    in  Map.union prefixedD1 d2
    decompose dc = let unPrefixedD1 = removePrefixFromValues sysName dc
                   in (unPrefixedD1, dc)
    checkComp d1 d2 = let prefixedD1 = addPrefixToValues sysName elementNames d1
```

```
                      in prefixedD1 == d2 || (Map.size (Map.intersection␣
↪prefixedD1 d2) == 0)
    extract1 dc = removePrefixFromValues sysName dc
    extract2 dc = dc
    augment1 dc d1 = addPrefixToValues sysName elementNames d1
    augment2 dc d2 = d2
```

```
Line 2: Eta reduce
Found:
syncToExistingSystem (sysName, systemSpec) system
  = syncEventSystems domainComposition system1 system
Why not:
syncToExistingSystem (sysName, systemSpec)
  = syncEventSystems domainComposition system1Line 15: Eta reduce
Found:
extract1 dc = removePrefixFromValues sysName dc
Why not:
extract1 = removePrefixFromValues sysNameLine 17: Eta reduce
Found:
augment1 dc d1 = addPrefixToValues sysName elementNames d1
Why not:
augment1 dc = addPrefixToValues sysName elementNames
```

`syncSpecList` takes a list of named system specifications and returns the resulting synchronized system.

[43]:
```
syncSpecList :: [(String, SystemSpecification)] -> System DomainValue
syncSpecList [] = error "module should at least have 1 system specification"
syncSpecList [(name, systemSpec)] = declareSystem systemSpec
syncSpecList [namedSystemSpec1, namedSystemSpec2] = syncTwoSystemSpecs␣
 ↪namedSystemSpec1 namedSystemSpec2
syncSpecList (namedSystemSpec:rest) = syncToExistingSystem namedSystemSpec␣
 ↪(syncSpecList rest)
```

For defining restrictions, it is useful to refer to states and events of the system by the name of the subsystem they belong to. The function `renameStatesAndEvents` adds a prefix to all states and events labels.

[44]:
```
addPrefixToRelation :: String -> RelationDeclaration -> RelationDeclaration
addPrefixToRelation prefix (Relation n r) = Relation (prefix ++ n) r
addPrefixToRelation prefix (Guard n r) = Guard (prefix ++ n) r
addPrefixToRelation prefix (Function n r) = Function (prefix ++ n) r

renameStatesAndEvents :: String -> SystemSpecification -> SystemSpecification
renameStatesAndEvents
    prefix
```

```haskell
  (SystemSpecification domainElements controllableEvents uncontrollableEvents␣
↪otherOps machine)
 = SystemSpecification domainElements controllableEvents'␣
↪uncontrollableEvents' otherOps machine'
 where
  addPrefix = ((prefix ++ ".")++)
  addPrefixes = map (addPrefixToRelation (prefix ++ "."))
  controllableEvents' = addPrefixes controllableEvents
  uncontrollableEvents' = addPrefixes uncontrollableEvents
  eventNames = map f $ controllableEvents ++ uncontrollableEvents
    where
      f (Relation n r) = n
      f (Guard n r) = n
      f (Function n r) = n

  machine' = map f machine
    where
      f (Edge "from" origin "to" target "with" (r1name, "id", r2name))
        =
         Edge "from" (addPrefix origin) "to" (addPrefix target) "with"␣
↪(r1name, "id", r2name)
      f (Edge "from" origin "to" target "with" (r1name, ename, r2name))
        | ename `elem` eventNames =
         Edge "from" (addPrefix origin) "to" (addPrefix target) "with"␣
↪(r1name, addPrefix ename, r2name)
        | otherwise =
         Edge "from" (addPrefix origin) "to" (addPrefix target) "with"␣
↪(r1name, ename, r2name)
      f (State l) = State $ addPrefix l
      f (InitialState l) = InitialState $ addPrefix l
      f (TerminalState l) = TerminalState $ addPrefix l
      f (InitialTerminalState l) = InitialTerminalState $ addPrefix l
```

`getRestrictions` takes a system, an event label, and a state label, and returns the corresponding restriction list.

[45]:
```haskell
checkStateLabel :: String -> StateLabel -> Bool
checkStateLabel label (SingleLabel stateLabel) = label == stateLabel
checkStateLabel label (JointLabel sl1 sl2) = checkStateLabel label sl1 ||␣
↪checkStateLabel label sl2

getRestrictions :: System d -> String -> String -> [Restriction d]
getRestrictions (System machine contrs uncontrs dom initD) eventLabel stateLabel
 = case event of
    Just ev -> [(ev, st) | st <- states]
    Nothing -> error $ "event " ++ eventLabel ++ "not found"
 where
```

```
    (Automaton qs is ts tp delta) = machine
    states = filter (checkStateLabel stateLabel) qs
    event = find (\e -> getLabel e == eventLabel) (contrs ++ uncontrs)
```

declareModule is then the valuation function for the ModuleSpecificication syntax.

[46]:
```
declareModule :: ModuleSpecification -> System DomainValue
declareModule moduleDeclarations = system \/ restrictions
    where
        transformedEvents = (\(c,u) -> (Map.unions c, Map.unions u)) $ unzip $␣
↪map f moduleDeclarations
          where
            f (DeclareSystem sysName systemSpecification) =␣
↪getTransformedEvents (sysName, systemSpecification)
            g (sysName, eventName) = addPrefixToRelation (sysName ++ ".")

        systemSpecs = map f $ filter g moduleDeclarations
          where
            f (DeclareSystem sysName systemSpecification) = (sysName,␣
↪applyEventRefs (renameStatesAndEvents sysName systemSpecification)␣
↪transformedEvents)
            g DeclareSystem {} = True
            g _ = False

        system = syncSpecList systemSpecs

        restrictions = concatMap f $ filter g moduleDeclarations
        g DeclareRestriction {} = True
        g _ = False
        f (DeclareRestriction state "restricts" event) = getRestrictions system␣
↪event state
```

What follows is a example module, containing an actuator, a sensor, and a requirement for the behavior of the resulting system.

[47]:
```
exampleModule = [
    DeclareSystem "actuator" SystemSpecification
        { domainElements = [
              BoolElement "actuatorStatus" False
          ],
          controllableEvents = [
              Function "switchActuator" $ do {
                  actuatorStatus <- getBoolValue "actuatorStatus";
                  setBoolValue "actuatorStatus" (not actuatorStatus);
                  }
          ],
          uncontrollableEvents = [],
```

```
        otherOperations = [],
        machine = [
            InitialTerminalState "actuatorOff",
            State "actuatorOn",

            Edge from "actuatorOff" to "actuatorOn" with ("id",␣
↪"switchActuator", "id"),
            Edge from "actuatorOn" to "actuatorOff" with ("id",␣
↪"switchActuator", "id")
        ]
      },

  DeclareSystem "sensor" SystemSpecification
      { domainElements = [
            BoolElement "sensorStatus" False
        ],
        controllableEvents = [],
        uncontrollableEvents = [
            Function "switchSensor" $ do {
                sensorStatus <- getBoolValue "sensorStatus";
                setBoolValue "sensorStatus" (not sensorStatus);
                }
        ],
        otherOperations = [],
        machine = [
            InitialTerminalState "sensorOff",
            State "sensorOn",

            Edge from "sensorOff" to "sensorOn" with ("id", "switchSensor",␣
↪"id"),
            Edge from "sensorOn" to "sensorOff" with ("id", "switchSensor",␣
↪"id")
        ]
      },

  DeclareSystem "requirement" SystemSpecification
      { domainElements = [],
        controllableEvents = [],
        uncontrollableEvents = [],
        otherOperations = [],
        machine = [
            InitialTerminalState "r0",
            State "r1",

            Edge from "r0" to "r1" with ("id", "actuator.switchActuator",␣
↪"id"),
            Edge from "r1" to "r0" with ("id", "sensor.switchSensor", "id")
```

```
        ]
      }
    ]
```

[48]:
```
exampleModuleSystem = declareModule exampleModule

print exampleModuleSystem
```

```
Controllable Events: [actuator.switchActuator]
Uncontrollable Events: [sensor.switchSensor]
Domain: [fromList [("actuator.actuatorStatus",True),("sensor.sensorStatus",True)],fromList [("a
Initial values: [fromList [("actuator.actuatorStatus",False),("sensor.sensorStatus",False)]]

Machine:
States: [(actuator.actuatorOff,sensor.sensorOff,requirement.r0),(actuator.actuatorOff,sensor.se
Initial States: [(actuator.actuatorOff,sensor.sensorOff,requirement.r0)]
Terminal States: [(actuator.actuatorOff,sensor.sensorOff,requirement.r0)]
Alphabet: [
    (id,actuator.switchActuator,id),
    (id,sensor.switchSensor,id),
]

Transitions:
(actuator.actuatorOff,sensor.sensorOff,requirement.r0) ==> (id,actuator.switchActuator,id) = [

(actuator.actuatorOff,sensor.sensorOff,requirement.r1) ==> (id,sensor.switchSensor,id) = [(actu

(actuator.actuatorOff,sensor.sensorOn,requirement.r0) ==> (id,actuator.switchActuator,id) = [(a

(actuator.actuatorOff,sensor.sensorOn,requirement.r1) ==> (id,sensor.switchSensor,id) = [(actua

(actuator.actuatorOn,sensor.sensorOff,requirement.r0) ==> (id,actuator.switchActuator,id) = [(a

(actuator.actuatorOn,sensor.sensorOff,requirement.r1) ==> (id,sensor.switchSensor,id) = [(actua

(actuator.actuatorOn,sensor.sensorOn,requirement.r0) ==> (id,actuator.switchActuator,id) = [(ac

(actuator.actuatorOn,sensor.sensorOn,requirement.r1) ==> (id,sensor.switchSensor,id) = [(actuat
```

[49]:
```
take 2 $ getTraces exampleModuleSystem
```

```
[([],fromList [("actuator.actuatorStatus",False),("sensor.sensorStatus",False)]),([actuator.sw
```

We define another module with an actuator and a sensor where the `sensorSwitch` event is restricted by the state `actuatorOff` of the actuator.

[50]:
```
exampleModule2 = [
    DeclareSystem "actuator" SystemSpecification
        { domainElements = [
              BoolElement "actuatorStatus" False
          ],
          controllableEvents = [
              Function "switchActuator" $ do {
                  actuatorStatus <- getBoolValue "actuatorStatus";
                  setBoolValue "actuatorStatus" (not actuatorStatus);
                  }
          ],
          uncontrollableEvents = [],
          otherOperations = [],
          machine = [
              InitialTerminalState "actuatorOff",
              State "actuatorOn",

              Edge from "actuatorOff" to "actuatorOn" with ("id",␣
↪"switchActuator", "id"),
              Edge from "actuatorOn" to "actuatorOff" with ("id",␣
↪"switchActuator", "id")
          ]
        },

    DeclareSystem "sensor" SystemSpecification
        { domainElements = [
              BoolElement "sensorStatus" False
          ],
          controllableEvents = [],
          uncontrollableEvents = [
              Function "switchSensor" $ do {
                  sensorStatus <- getBoolValue "sensorStatus";
                  setBoolValue "sensorStatus" (not sensorStatus);
                  }
          ],
          otherOperations = [],
          machine = [
              InitialTerminalState "sensorOff",
              State "sensorOn",

              Edge from "sensorOff" to "sensorOn" with ("id", "switchSensor",␣
↪"id"),
              Edge from "sensorOn" to "sensorOff" with ("id", "switchSensor",␣
↪"id")
          ]
        },
```

```
    DeclareRestriction "actuator.actuatorOff" restricts "sensor.switchSensor"
    ]
```

[51]:
```
exampleModuleSystem2 = declareModule exampleModule2

print exampleModuleSystem2
```

```
Controllable Events: [actuator.switchActuator]
Uncontrollable Events: [sensor.switchSensor]
Domain: [fromList [("actuator.actuatorStatus",True),("sensor.sensorStatus",True)],fromList [("a
Initial values: [fromList [("actuator.actuatorStatus",False),("sensor.sensorStatus",False)]]

Machine:
States: [(actuator.actuatorOff,sensor.sensorOff),(actuator.actuatorOff,sensor.sensorOn),(actuat
Initial States: [(actuator.actuatorOff,sensor.sensorOff)]
Terminal States: [(actuator.actuatorOff,sensor.sensorOff)]
Alphabet: [
    (id,actuator.switchActuator,id),
    (id,sensor.switchSensor,id),
]

Transitions:
(actuator.actuatorOff,sensor.sensorOff) ==> (id,actuator.switchActuator,id) = [(actuator.actuat

(actuator.actuatorOff,sensor.sensorOn) ==> (id,actuator.switchActuator,id) = [(actuator.actuato

(actuator.actuatorOn,sensor.sensorOff) ==> (id,actuator.switchActuator,id) = [(actuator.actuato

(actuator.actuatorOn,sensor.sensorOff) ==> (id,sensor.switchSensor,id) = [(actuator.actuatorOn

(actuator.actuatorOn,sensor.sensorOn) ==> (id,actuator.switchActuator,id) = [(actuator.actuator

(actuator.actuatorOn,sensor.sensorOn) ==> (id,sensor.switchSensor,id) = [(actuator.actuatorOn,s
```

Appendix E

Simulator

This appendix contains some documentation regarding the simulator for X-Control.

```
module Simulator (
    SimulatorCommand(UncontrollableEvent, ClockTick, Quit),  simulateSystem,
    commandListener,  getCommand,  clock,  executeSimulation,
    selectControllableEvent,  selectUncontrollableEvent,  getRandomListElement,
    getEnabledEvents,  getEnabledTransitions
  ) where
```

`data SimulatorCommand`

The commands that can be given to the simulator.

Constructors

=	`Quit`	This command stops the simulator.
|	`ClockTick`	This is a clock tick command, which initiates a controllable event.
|	`UncontrollableEvent`	This command initiates a user selected uncontrollable event.

`simulateSystem :: (Show d, Eq d) => System d -> IO ()`

Simulates a given System. Initiates `commandListener` and `clock` on two seperate threads.

`commandListener :: Lock -> MVar SimulatorCommand -> IO ()`

Retrieves commands from user, and puts them in `commandVar`. Must acquire lock before listening to input stream, only releases lock when a non-valid command code is retrieved from the input stream.

```
getCommand :: Char -> Maybe SimulatorCommand
```

Obtain command from character code.

```
clock :: MVar SimulatorCommand -> IO ()
```

Generated ClockTick commands every second. If the commandVar is not empty, then the tick is skipped.

```
executeSimulation :: (Show d, Eq d) =>
                     System d
                     -> MVar SimulatorCommand -> Lock -> StateLabel -> d -> IO ()
```

Executes the simulation step given the system, the command variable, the commandlock, the current state, and the current data variable. Retrieves command from either the commandListener or the clock. Re-runs the simulator after a command is recieved and handled, unless the Quit command is given.

```
selectControllableEvent :: Eq d =>
                           System d -> StateLabel -> d -> IO (StateLabel, d)
```

Given a system, a current state, and a current value, randomly select a next state and value retrieved by taking a transistion with a controllable event from the current state and value.

```
selectUncontrollableEvent :: (Eq d, Show d) =>
                             System d -> StateLabel -> d -> IO (StateLabel, d)
```

Given a system, a current state, and a current value, let the user select a next state and value retrieved by taking a transition with a uncontrollable event from the current state and value.

```
getRandomListElement :: [a] -> IO a
```

Given a list of items, return a randomly selected item.

```
getEnabledEvents :: Eq d =>
                    EventMachine d -> [Event d] -> StateLabel -> d -> [Event d]
```

Given an EventMachine, a list of events, a current state, and a current value, returns all events from the list which are enabled in the current state with the current value.

```
getEnabledTransitions :: Eq d =>
                         EventMachine d -> [Event d] -> StateLabel -> d -> [(StateLabel, d)]
```

Given an EventMachine, a list of events, a current state, and a current value, returns all new state and value pairs obtained from transitions wich have events from the list, and are enabled in the current state with the current value.