Behavioral Specification of Reactive Systems Using Stream-Based I/O Tables

Benjamin Hummel Judith Thyssen
Technische Universität München
Institut für Informatik
Munich, Germany
{hummelb,thyssen}@in.tum.de

***Abstract*—A core problem in formal methods is the transition from informal requirements to formal specifications. Especially when specifying reactive systems, many formalisms require the user to either understand a complex mathematical theory and notation or to derive details not given in the requirements, such as the state space of the problem. While formalizing a real-world requirements document, we developed a technique where not states but signal patterns are the main elements. We argue that it supports a formalization that is often closer to the informal requirements and thus provides a smoother transition to formal methods. As only tables of regular expressions are used for notation, the technique can easily be understood by non-mathematicians. Many properties, such as consistency, can be checked automatically on these specifications. Besides the formal foundation of our approach, this paper presents prototypical tool support and first results from an industrial case study.**

***Index Terms*—tabular specification; consistency; streams**

I. Introduction

The goal of this paper is to present a specification technique for reactive systems that is close enough to textual requirement documents and simple enough to be understandable and usable by domain experts, which are not necessarily willing to learn complicated specification formalisms. Our technique eases the transition from textual requirements to a formal and analyzable specification. It was developed during the formalization of a textual requirements specification provided by Siemens, which describes a machine from the automation domain.

Textual requirements are usually either formulated as examples (scenarios) or as rules which the system has to follow. The latter are either formulated in terms of states ("if the system is in state X and Y happens, then") or observations on sequences of input and output events ("after message A has been received four times, message B should be sent at least twice"). While there is ample amount of techniques for the formalization of scenarios (such as message sequence charts [1] or UML sequence diagrams [2]) or state based rules (such as state charts [3], [4]), we think there is a lack of easily applicable techniques for sequence oriented requirements. This paper indicates a possible direction for closing this gap.

Parts of this work have been conducted in a project with Siemens Sector Industry.

Scope: A common approach used for the specification of embedded reactive systems is based on the decomposition of the system into hierarchical components whose interface is described by typed ports which are connected by channels. Components are either described as composition of other components or by a relation on (usually infinite) sequences of input and output messages. Examples of this are the asynchronous FOCUS [5] or the synchronous Lustre [6].

While engineers usually easily understand the component decomposition, the specification of primitive components is often still a challenge. Reasons for this include that usually the formalisms are either based on mathematical notations which need expertise to apply, or require a non-trivial transformation step from several isolated requirements – usually formulated as observations of the system – to a consistent description which incorporates additional information such as the structure of the internal state space of the component. Because of this we consider a system decomposition to be given and concentrate on the specification of the behavior of a single component.

Contribution: In this paper we propose a formalism based on regular and ω-regular languages and a complementing notation based on regular expressions and tables usable for the description of functions on streams. In conjunction with the decomposition approaches mentioned before it can be used for the description of reactive systems. Since the used patterns are closely related to informal textual descriptions, our approach allows a straightforward translation of textual requirement documents to formal specifications in many cases. This is further supported by the use of simple concepts of tables and regular expressions, which allow engineers to familiarize themselves with the technique more easily. All in all, our approach results in comprehensive specifications that can be understood and developed by requirements engineers or customers since not the same profound mathematical knowledge is needed as for example for logical or automata-based specifications. Additionally, all relevant properties of these specifications (*e. g.,* input completeness or consistency) can be checked automatically, forming the basis for prototypical tool support.

Outline: The paper is structured as follows: Sec. II informally introduces our table-based specification technique by means of an example from the automation domain. In Sec. III, we relate our approach to existing specification

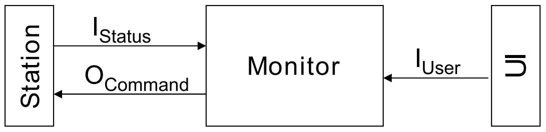


Fig. 1. Syntactic Interface of the Monitor

techniques before we present its formal semantics in Sec. IV. Based on these definitions, Sec. V explains the analysis of the specification for different properties. Sec. VI describes our prototypical tool implementation and first results from a case study. There we also discuss the benefits and limitations of our approach before concluding the paper in Sec. VII.

II. Motivating Example

Before we define the syntax and semantics of stream-based I/O tables in detail in Sec. IV, we illustrate the concepts by means of a simple example from the automation domain. We specify a monitoring component (just called *monitor* in the remainder) via which a single station can be logged in and out of a system. The monitor is used to decouple the user interface from the external station. In the following paragraphs, firstly the requirements are informally described, secondly the syntactic interface, *i. e.,* the input and output ports through which the monitoring station is connected with its environment, is identified, and lastly the requirements are formally specified and analyzed by means of stream-based I/O-tables based on regular expressions as introduced in more detail in Sec. IV.

Informal Description: The monitor controls the log in and out of a certain station based on user input. Besides, it continuously checks the status of the station. The monitor must fulfill the following requirements:

1) The user can send a request to the monitor demanding to log the station in or out. In the next time interval the monitor will send a corresponding signal to the station if the station is not already logged in resp. out.
2) The station sends its current status (on or off) to the monitor within regular intervals. The monitor acknowledges the receiving of a status signal within the next three time intervals.
3) If no status signal from the station is received for five time intervals, the monitor will send a status request to the station.

Syntactic Interface: Based on the given informal requirements, we identify the input and output ports through which the monitor is connected with its environment. Fig. 1 schematically depicts the syntactic interface of the monitor. The monitor has two input ports, I_{User} to receive the user input and I_{Status} to receive the status of the station, and one output port O_{Command} via which the monitor sends commands to the station. The respective types of the ports are given in Tab. I. The empty message ϵ is explicitly included in the types in order to be able to model missing interaction on a port in a certain time interval.

Table-based Specification: Having defined the syntactic interface, we build up the table-based specification shown in Tab. II. The input and output ports define the columns of the table, while the different rows of the table specify the different requirements on the I/O behavior of the monitor. The expressions in the input cells define input patterns by describing the last messages that have been received on the input ports until a time t. The expressions in the output cells define corresponding output patterns, *i. e.,* messages that must be sent in the following time intervals whenever the input patterns are fulfilled. In order to preserve the mapping between the informal requirements and the formal specification, the table comprises an additional column for annotations. The requirements are grouped into two different segments separated by a double horizontal line: The first segment formalizes the logging in/out of the station as described in Requirement 1. The second segment formalizes the status control as described in Requirements 2 and 3. Different segments must always be fulfilled simultaneously. The mapping from requirements to table segments is not necessarily one to one, but rather a design decision.

To clarify the presented table-based specification, we explain some of the expressions used in the table. The regular expression "$\mathsf{on}\ \epsilon^*$" models that the last signal received on port I_{Status} until time t has been on. The time intervals after that, only ϵ – meaning no signal – has been received. The notation $x\{min, max\}$ is used to describe that a certain signal or regular expression x occurs min to max times in a row. Thus, the expression "$\epsilon\{5, 5\}$" formalizes that in the last five time intervals no signal is received on the respective port. The character $\mathsf{.}$ declares that any not further specified signal of the corresponding type (including ϵ) is received/sent in the respective time interval. The expression "$\mathsf{.}\{0, 2\}\ \mathsf{ack}$" describes that – starting from time interval $t + 1$ – first for 0 to 2 time intervals any signal can be sent, but at latest in the third time interval, ack has to be sent.

Analysis of the Specification: So far, the presented specification exactly reflects the informal requirements. In this paragraph, we analyze the specification for different properties, namely input completeness and consistency. Input completeness means that there is no input for which no output reaction is defined. The analysis for input completeness allows for detecting underspecification in the requirements. Consistency of the specification means that the different requirements do not impose contradicting output reactions.

The specification given in Tab. II is obviously not input complete. For example, in the first segment it is undefined what has to happen if the user sends a command but the station did not yet send its status ($I_{\text{User}} = \epsilon^k\mathsf{on}$, $I_{\text{Status}} = \epsilon^{k+1}$) or if the user gives no command at all. While it is trivial to see in this example, for more complicated cases tool support is inevitable.

The specification is not consistent either. For example, the second segment requires the monitor to send a req to the station if no status has been received for some time. If the user at the same time presses the on button, the first segment

TABLE I
TYPES OF THE PORTS OF THE SYNTACTIC INTERFACE

Port	*Type*	*Description*
I_{User}	$\{\mathsf{on}, \mathsf{off}, \epsilon\}$	User request: log station in (on), log station out (off), or no user input (ϵ)
I_{Status}	$\{\mathsf{on}, \mathsf{off}, \epsilon\}$	Status report: logged in (on), logged out (off), or no status signal (ϵ)
O_{Command}	$\{\mathsf{on}, \mathsf{off}, \mathsf{ack}, \mathsf{req}, \epsilon\}$	Command to station: log in (on), log out (off), acknowledgment of a status signal (ack), request of a status signal (req), or no signal (ϵ)

TABLE II
INITIAL I/O TABLE FOR THE MONITOR

I_{User}	I_{Status}	O_{Command}	*Annotation*
on	off ϵ^*	on	Requirement 1: When the user input on is received and the last status signal has been off, the monitor sends the command on to the station.
off	on ϵ^*	off	Requirement 1: When the user input off is received and the last status signal has been on, the monitor sends the command off to the station.
.	$\epsilon\{5,5\}$	req	Requirement 3: When the monitor receives 5 time intervals no status signal (ϵ), it sends a request (req) to the station.
.	on|off	$.\{0,2\}$ ack	Requirement 2: When the station signals a status (on or off), the monitor has to confirm this (ack) within the next three time intervals.

requires to send the on command instead, which contradicts the action required by the second segment.

III. RELATED WORK

Before we define the semantics of our table-based specification technique in detail, we relate our approach to existing work. The number of formalisms for the specification of reactive systems, including all of their extensions and alternative semantics, prohibits a complete enumeration. Instead we give a classification of approaches and discuss only the most prominent examples. The focus is especially on the transition from textual requirements to formal specifications.

Linguistic Approaches: One way to ease this transition is to restrict the language used in textual requirements and thus reduce the gap to formal specifications. One example are informal or semi-formal approaches, such as [7], which aim at improving *textual* requirement documents for human readers by providing rules on word order and the structure of the sentences used. While these approaches help in writing clear and concise requirements documents, they are no substitute for formal specifications when formal methods are to be applied.

Specification Techniques: The complementary approach for making the formalization more easy is to use a specification formalism that is easy enough to allow specifications to be understood or even created by the domain experts, which are also responsible for the requirements. Specification techniques are usually either state-based or stream-based. The former class describes the system behavior using some variant of automaton or state chart, which defines the output depending on the input and the current state [3], [4]. These techniques are a good match if the requirements are written down in a state oriented way. Though requirements focusing on sequences of input messages can be captured using state-based specifications, the transition to a state-based formalism usually is a non-trivial step. The transition includes the definition of a suitable state space which often is rather seen as an activity of the detailed specification or implementation phase.

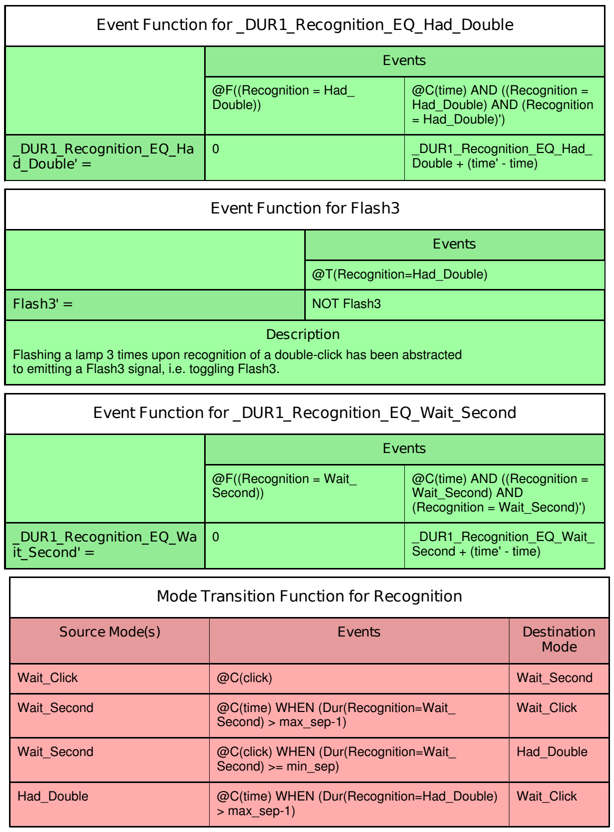
So for requirements described in terms of input and output sequences, specifications based on streams often are a better match. They only describe the externally visible (black-box) behavior. Therefore the system is interpreted as a relation on sequences (streams) of inputs and outputs and a specification is a predicate which restricts the valid pairs of I/O streams [5]. One way of describing these predicates is to use some kind of mathematical logic, which is difficult to use for practitioners. To hide the complexity of these formalisms, several aids have been proposed. For example, Dwyer et al. [8] provide a catalog of common patterns used in temporal logic. Bauer et al. [9] describe the language SALT, which is an advanced syntax for temporal logic and can be compiled to (T)LTL. Both approaches reduce common sources of errors in specifications written in temporal logic, but still require profound knowledge of logics.

Another widely accepted technique for the formalization of interaction sequences especially in the requirements capture phase of the software development process are message sequence charts/sequence diagrams [1], [2]. However, they are mostly used for the specification of exemplary sequences (scenarios) and thus do not result in a complete description of the system behavior, which we are aiming at, but merely express a partial execution trace. Live sequence charts [10], [11] try to bridge the gap from existential scenarios to universal specifications and introduce precharts, which are similar to the input part of our table rows. The same holds for conditional scenarios and their triggers in triggered message sequence charts [12]. Our approach differs from these MSC variants, in that we focus on the specification of single components (as compared to the interaction between different components). Furthermore our goal was to find a fairly reduced set of description elements and a compact notation to ease the understanding for domain engineers, who often do not have a computer science background and are easily overwhelmed by the possibilities of MSCs and their variants.

Tabular Notations: A special class of specification techniques are those based on tabular notations. The usual claim is that tables are a familiar notation and allow for more clarity and structure and thus simplify the application of those techniques. While we use a tabular notation in this paper for the same reason, contrary to our work, to the best of our knowledge, all existing approaches are state-based, *i. e.,* a user of the technique has to divide the state space of the problem domain in a suitable way. The most well-known specification technique employing tables is the *software cost reduction* method (SCR) [13] which was developed based on the work of Parnas [14]. SCR expresses complex temporal correlations using different modes and mode switches, which basically can be interpreted as states. While these techniques have been applied successfully [15] and also large amount of work has been spent on semantics [16] and tool support [17], [18] for these tables, there are cases where making the state space explicit is not only non-trivial, but also leads to larger and more complicated specifications. An example for this case is given at the end of this section. Parnas himself provides a collection of table types and abbreviation strategies for organizing and simplifying functional and relational expressions [19]. In contrast to our approach, he aims at providing optimal tables for each problem. However, the need to understand the resulting, quite difficult table semantics as well as the fact that the table entries must be formulated as logical formulas, still requires profound mathematical knowledge. Besides these two there are further tabular notations for state machines summarized in [20].

The Double-Click Detector. We want to conclude this section by giving a simple (albeit slightly artificial) example which clearly shows the differences of our approach to other tabular specification techniques, such as SCR. The examined system has a single input, which signals for each time tick whether a button has been clicked or not, and a single output, which is connected to a light and is used to switch it on and off. The goal is to detect double clicks and acknowledge it to the user by light signals. More concretely: If two clicks (true at input) are found with at least one and at most three time intervals in between, this is called a double click. If a sequence of more than two clicks with the above mentioned delay between consecutive pairs is found, only the first pair is recognized as a double click (no triple clicks). A recognized double click is acknowledged by flashing the light three times.

The formalization of these requirements using SCR is shown in Fig. 2[1] while the solution using I/O tables is depicted in Tab. III. Note that in the SCR example the last table is the most important one, as it defines the mode transitions, while the other tables basically describe the setup for the detector. So this is the part, which should be compared to our table, which formalizes the requirements using only one non-trivial row. In contrast, SCR requires the introduction of multiple modes (*Wait_Click*, *Wait_Second*), which were not given in the original requirements, even though the requirement of flashing three times was completely abstracted away. Of course there are many cases as well, where SCR has its advantages (especially if the modes are already mentioned in the requirements). Hence, the choice of the "right" technique depends on the requirements to be formalized.



Event Function for _DUR1_Recognition_EQ_Had_Double

	Events	
	@F((Recognition = Had_ Double))	@C(time) AND ((Recognition = Had_Double) AND (Recognition = Had_Double)')
_DUR1_Recognition_EQ_Ha d_Double' =	0	_DUR1_Recognition_EQ_Had_ Double + (time' - time)

Event Function for Flash3

	Events
	@T(Recognition=Had_Double)
Flash3' =	NOT Flash3

Description

Flashing a lamp 3 times upon recognition of a double-click has been abstracted to emitting a Flash3 signal, i.e. toggling Flash3.

Event Function for _DUR1_Recognition_EQ_Wait_Second

	Events	
	@F((Recognition = Wait_ Second))	@C(time) AND ((Recognition = Wait_Second) AND (Recognition = Wait_Second)')
_DUR1_Recognition_EQ_Wa it_Second' =	0	_DUR1_Recognition_EQ_Wait_ Second + (time' - time)

Mode Transition Function for Recognition

Source Mode(s)	Events	Destination Mode
Wait_Click	@C(click)	Wait_Second
Wait_Second	@C(time) WHEN (Dur(Recognition=Wait_ Second) > max_sep-1)	Wait_Click
Wait_Second	@C(click) WHEN (Dur(Recognition=Wait_ Second) >= min_sep)	Had_Double
Had_Double	@C(time) WHEN (Dur(Recognition=Had_Double) > max_sep-1)	Wait_Click

Fig. 2. Double click detection using SCR.

IV. Structure and Semantics

Having already presented an example in Sec. II, we give the formal syntax and semantics for stream-based I/O tables in this section. We start with the general structure of such a table (abstract syntax) followed by its semantic interpretation. This section is closed by proposing a concrete syntax based on POSIX regular expressions.

Before, we have to introduce some notation. For an alphabet (set) Σ we denote the set of finite sequences over Σ by Σ^* and the set of infinite sequences by Σ^ω, where an infinite sequence is just a mapping $\mathbb{N} \to \Sigma$. We call those infinite sequences *streams*[2]. We are using streams for modeling communication over time by assuming time to be split into intervals which can be counted by $\mathbb{N}$. A stream associates a transmitted message with each of those intervals. For a stream s we denote its prefix of length t by $s{\downarrow}t$ and its (infinite) suffix after removing the first t elements by $s{\uparrow}t$. Furthermore, we use $s \frown s'$ to denote the concatenation of a sequence s and a stream s'.

[1]We want to thank Ralph Jeffords and Constance Heitmeyer for providing us with the SCR formalization, which contains 6 additional dictionary tables not shown here.

[2]In Lustre these would be called flows.

TABLE III
DOUBLE CLICK DETECTION USING STREAM-BASED I/O TABLES.

I_{in}	O_{out}
false{4,4} true false{1,3} true	false (true false){3,3}
.*	.*

A. Abstract Syntax

Following [5] we describe the syntactic interface of a component or a system by the ports connecting it to the environment. A set P of port identifiers is called a *typed port set*, if there is a mapping type from those ports to finite sets[3]. The syntactic interface is then described by two typed port sets P_I and P_O describing the input and output, denoted by $(P_I \blacktriangleright P_O)$.

A syntactic interface $(P_I \blacktriangleright P_O)$ with $P_I = \{i_1, ...i_n\}$ and $P_O = \{o_1, ..., o_m\}$ introduces alphabets $\Sigma_I = (\mathsf{type}(i_1) \times ... \times \mathsf{type}(i_n))$ of input messages and $\Sigma_O = (\mathsf{type}(o_1) \times ... \times \mathsf{type}(o_m))$ of output messages. The alphabet of input/output pairs is written as $\Sigma_{I/O} = \Sigma_I \times \Sigma_O$. In this framework we can specify components by relations on input and output streams or more directly as a subset of $\Sigma^\omega_{I/O}$. The tables introduced in this paper are just a means of specifying these sets.

Definition 1. *For a syntactic interface $(P_I \blacktriangleright P_O)$ a* stream-based I/O table *$\mathcal{T}$ is a set of segments, where each segment $S \in \mathcal{T}$ is a tuple (I, O, r) such that*

- *r is the size of the segment (the number of* rows*),*
- *I is a sequence of length r of regular languages over the input alphabet Σ_I (*i. e., *$I \in (\mathcal{P}(\Sigma_I^*))^r$),*
- *O is a sequence of length r of ω-regular languages over the output alphabet Σ_O (*i. e., *$O \in (\mathcal{P}(\Sigma_O^\omega))^r$).*

Each tuple (I_k, O_k) with $k \in \{1, \dots, r\}$, consisting of the k-th elements of I and O, corresponds to a row of S.

This definition captures the structure of the tables given in the example where a table consists of several segments (parallel requirements) which in turn consist of ordered rows (alternating requirements) relating inputs to outputs. We do not include the annotation column in the formalization, as it is only used for documentation. The limitation to (ω-)regular languages is a prerequisite needed to show decidability of input completeness and consistency in Sec. V.

B. Semantics

As indicated before, the semantic meaning of a component with syntactic interface $(P_I \blacktriangleright P_O)$ is described by its valid (infinite) input/output behaviors, a subset of $\Sigma^\omega_{I/O}$. This is equivalent to a relation between input streams (Σ^ω_I) and output streams (Σ^ω_O). The valid I/O streams for a table segment are described in the following definition.

Definition 2. *The* valid I/O behaviors *of a segment $S = (I, O, r)$ of a stream-based I/O table with syntactic interface $(P_I \blacktriangleright P_O)$ are described by the set $L(S)$ defined as*

$$\{(x, y) \in \Sigma^\omega_I \times \Sigma^\omega_O | \forall t \in \mathbb{N}, \forall k \in \{1, \dots, r\} : \\ x{\downarrow}t \in I_k \setminus \textstyle\bigcup_{j=1}^{k-1} I_j \Rightarrow y{\uparrow}t \in O_k\} \ .$$

The implication captures the idea that the row of a segment whose input pattern matches at a given time t, determines the output following afterwards. The set difference makes sure that at any time only the first matching row is in effect. As a table consists of multiple segments which restrict the valid inputs and outputs at the same time, its valid behaviors are just the intersection of the behaviors allowed by the segments.

Definition 3. *Let $\mathcal{T}$ be a stream-based I/O table with syntactic interface $(P_I \blacktriangleright P_O)$. Then the* valid I/O behaviors *of $\mathcal{T}$ are given by the set*

$$L(\mathcal{T}) = \bigcap_{S \in \mathcal{T}} L(S) \ .$$

An alternative interpretation of a row of the table is that it disallows all streams which can be composed of a prefix whose input part matches the "left side" of the row, and a suffix whose output part does *not* match the "right side" of the row. Thus, for a segment $S = (I, O, r)$ the k-th row limits the valid I/O behaviors to

$$\overline{(I_k \boxtimes \Sigma^*_O) \frown \overline{(\Sigma^\omega_I \boxtimes O_k)}} \ ,$$

where $\boxtimes$ denotes the building of pairs of equal length words, *i. e.,* $L_1 \boxtimes L_2 = \{(x, y) \in L_1 \times L_2 \mid |x| = |y|\}$, and $\overline{L}$ the complement of a language L. This observation leads to an alternative definition of the valid I/O behaviors of a table.

Corollary 1. *Let $\mathcal{T}$ be a stream-based I/O table with syntactic interface $(P_I \blacktriangleright P_O)$. Then*

$$L(\mathcal{T}) \ = \bigcap_{(I,O,r) \in \mathcal{T}} \bigcap_{k=1}^{r} \overline{((I_k \setminus \bigcup_{j=1}^{k-1} I_j) \boxtimes \Sigma^*_O) \frown \overline{(\Sigma^\omega_I \boxtimes O_k)}} \ .$$

From Def. 2 it can also be seen that for the I/O behaviors of our specifications the inputs encountered up to some time t only influence the output starting from time $t + 1$. Stated differently, this means that the output at any time t only depends on the input until time $t - 1$ and *not* on future input. This property for behaviors is known as *strong causality* and is a crucial prerequisite for composability of components in asynchronous models of embedded systems, such as FOCUS [5][4].

[3]Here we just identify a type with its carrier set. Additionally we require a message to be present at each port at every time interval. Thus to model missing interaction, the empty message ϵ has to be part of the type (resp. its carrier set).

[4]For synchronous models the semantics of the tables could be tweaked to yield weakly causal behaviors, where the output may also depend on the input read at the same time.

While this property seems quite natural, many formalisms, such as relational specifications, easily allow the specification of systems which by accident know the future.

C. Concrete Syntax

From a theoretician's view the previous two sections covered everything relevant for the proposed specification technique. For practical purposes the concrete representation used for writing and manipulating specifications is of paramount importance. In fact, the described technique is rooted in its table-based representation, while the semantics has been detailed afterwards.

An example of our notation has already been given in Sec. II, so we only flesh out the details here. The primary structure is a table, where the columns of the table correspond to the ports of the component and the rows define requirements on the behavior. The columns are grouped into an input part (comprising the input ports) and an output part (comprising the output ports). The rows of the table are grouped into different segments (separated by double horizontal lines), which have to be respected in parallel. The order of rows within the segments determines their priorities, *i. e.,* for each time only the first row in a segment whose input part matches the input read so far constrains the output. Alternatively, this priority could have been abandoned and instead input languages within a segment could have been required to be disjunct. However, this priority arrangement of requirements within a segment allows for a more comfortable way of specifying the behavior of components, as for example exceptions can be specified in the first rows and must not be considered in the following rows specifying the normal behavior. Requiring disjunct input languages often results in very complex patterns and consequently contradicts our aim of obtaining a specification that is easy to write and understand.

Within each cell of the table is a regular expression using the alphabet defined by the type of the corresponding column's port. We follow the syntax of the POSIX extended regular expressions here, as this is well-known to many engineers from tools like *egrep* or search boxes in text editors and thus requires only little learning effort from its users.

For the interpretation of those regular expressions, let the syntactic interface of a specification be $((i_1, \dots, i_n) \blacktriangleright (o_1, \dots, o_m))$ and $\rho_1, \dots, \rho_n, \sigma_1, \dots, \sigma_m$ the regular languages described by the expressions of the k-th row. The input language I_k is

$$(\mathsf{type}(i_1)^* \frown \rho_1) \boxtimes \cdots \boxtimes (\mathsf{type}(i_n)^* \frown \rho_n) \ ,$$

i. e., the currently read input is matched "from the right". For the output language O_k we use

$$(\sigma_1 \frown \mathsf{type}(o_1)^\omega) \boxtimes \cdots \boxtimes (\sigma_m \frown \mathsf{type}(o_m)^\omega) \ ,$$

i. e., the future output has to be matched "from the left". Obviously, by this simple scheme we loose some expressive power, as we are not able to express infinite restrictions on the outputs and are limited in modeling dependencies between inputs from different ports. However, it turns out that even with these limitations many of the requirements discovered in practice can be formalized. Nonetheless, we are carefully looking into extensions of the concrete syntax which allow for more complex input and output languages as described in Sec. VI-C.

V. Analysis of I/O Tables

The formally funded specification of functional requirements by tables as shown in the previous section allows us to automatically check several properties of the specification, which would not be possible when working on requirement specifications written in natural language. Some of the properties described below, such as consistency, would not have to be checked when using other formalisms (*e. g.,* automata), but the transformation from requirements into these specifications requires more effort compared to the technique showed here. Thus our technique is more suitable for the early specification phase, as it allows for a smoother transition from requirements to formal models.

A. Input Completeness and Reachability

The first property we consider is *input completeness*, *i. e.,* whether a segment of the specification defines a reaction for every possible input. This usually indicates intentional or unintentional underspecification in the original requirements or an error during the formalization step. To decide whether this indicates an error or needs to be clarified by the stakeholders, it is important to find such undefined inputs.

Definition 4. *Let $S = (I, O, r)$ be a segment of a stream-based I/O table with syntactic interface $(P_I \blacktriangleright P_O)$. Then S is called* input complete*, iff $\bigcup_{k=1}^{r} I_k = \Sigma_I^*$.*

According to this definition a segment of a table-based specification is input complete if every possible input situation (*i. e.,* Σ_I^*) is covered by at least one input pattern of the different rows of the segments. The specification $\mathcal{T}$ is *input complete* iff all its segments $S \in \mathcal{T}$ are input complete.

Another flaw in the specification which can be recognized by looking only at the input is *reachability*, which means that certain parts in the specification are unused which again indicates errors in either the requirements or in their formalization. More precisely a row of a segment is unreachable if all inputs handled by this row have already been handled by the rows above.

Definition 5. *Let (I, O, r) be a segment of a stream-based I/O table with syntactic interface $(P_I \blacktriangleright P_O)$. Then we call the row $j \in \{1, \dots, r\}$* unreachable*, iff $I_j \setminus \bigcup_{k=1}^{j-1} I_k = \emptyset$.*

As I is a finite sequence of regular languages, the union and comparison operations can be calculated efficiently, only the complementation requires the conversion to deterministic automata which involves a potentially exponential blow-up. This leads to the following result.

Lemma 1. *Checking for input completeness and unreachable rows of a table segment is decidable. If the involved languages* *are given as regular expressions, both can be computed in exponential time.*

B. Consistency

As we are using constructs in our tables which are conceptionally nearer to requirements than to an implementation, we also inherit their drawbacks. In particular, this means that we can specify systems which are not realizable due to contradictions in the specification. There are two main sources for inconsistencies in our setup. One are inconsistencies within one section which are caused by output patterns restricting the future in a way not consistent with later actions required. This can only happen if the used output patterns make assumptions about more than one future time interval. The second source are inconsistencies in the reactions enforced by two parallel sections. We call a specification inconsistent, if there are inputs for which no output is valid according to the specification. Consistent specifications sometimes are also referred to as *input enabled*, as they can produce an output for any input sequence.

Definition 6. *Let $\mathcal{T}$ be a table-based I/O specification with syntactic interface $(P_I \blacktriangleright P_O)$ and $L(\mathcal{T})$ the language of all valid I/O histories. We call $\mathcal{T}$* consistent*, iff for each input history $i \in \Sigma_I^\omega$ there is at least one output history $o \in \Sigma_O^\omega$ with $(i, o) \in L(\mathcal{T})$. Otherwise $\mathcal{T}$ is called* inconsistent.

The next result demonstrates how consistency checking can be automated at least in theory.

Lemma 2. *Checking the consistency of a stream-based I/O table is decidable.*

Proof: Coroll. 1 gives a characterization of the valid I/O behaviors of a table using only set operations. All of the operations used are computable and closed for (ω-)regular languages [21], [22]. Hence, the consistency check is only a projection to the input alphabet, followed by complementation and an emptiness check. Thus checking for consistency is decidable. ■

Using the proof verbatim for implementing a consistency checker would result in triple exponential complexity. A more efficient solution is presented next.

C. Practical Consistency Checking for Safety Properties

The consistency check given before has two major drawbacks for practical purposes. Firstly, the complexity due to the repeated complementation makes it practically infeasible for any non-trivial specification. Secondly, the implementation of a complementation of ω-regular languages, while theoretically solved, is quite a challenge in practice. Here we introduce a construction based on deterministic finite automata (DFAs) which works if the specification contains only safety properties.

A *DFA* is a tuple $A = (\Sigma, Q, q^0, F, \delta)$ with finite alphabet Σ, state set Q, starting state $q^0 \in Q$, and final states $F \subseteq Q$. The total transition function $\delta : Q \times \Sigma \to Q$ is inductively extended to words by $\delta(q, uv) = \delta(\delta(q, u), v)$. A finite word w is *accepted* by a DFA, if $\delta(q^0, w) \in F$. The set of all words accepted by A is denoted by $L(A)$.

To use DFAs for this problem, we do not construct an automaton which accepts all valid I/O histories but rather finite prefixes of *invalid* histories. Thus a valid run of the system would try to navigate around the final states of the automaton. As violations of liveness properties are often only visible in infinite streams this solution only works for safety properties. Again we first deal with individual rows of the specification. The key idea is to “start” the automaton for invalid output whenever the input of the row is recognized. However, as the execution of the output may take multiple steps and we could match the input more than once in this time, we have to be able to “start” the automaton more than once as well. This can be achieved using the well-known power set construction.

Lemma 3. *Let (I_k, O_k) be a row of a stream-based I/O table with syntactic interface $(P_I \blacktriangleright P_O)$ and $A_I = (\Sigma_I, Q_I, q_I^0, F_I, \delta_I)$ and $A_O = (\Sigma_O, Q_O, q_O^0, F_O, \delta_O)$ DFAs such that $I_k = L(A_I)$ and $O_k = L(A_O) \frown \Sigma_O^\omega$. Then there is a DFA with at most $|Q_I| 2^{|Q_O|}$ states recognizing all finite violations for this row.*

Proof: We may assume the automaton A_O to be minimal and denote by $s_O \in Q_O$ its sink state, *i. e.,* the only state from which no final state is reachable. If no such sink state exists, A_O accepts any word and the row can be ignored. The resulting row automaton A_R is then given by $(\Sigma_I \times \Sigma_O, Q_I \times 2^{Q_O}, q_R^0, Q_I \times F_R, \delta_R)$, where $q_R^0 = (q_I^0, \emptyset)$ if $q_I^0 \notin F_I$ and $q_R^0 = (q_I^0, \{q_O^0\})$ otherwise, $F_R = \{f \subseteq Q_O \mid s_O \in f\}$, and δ_R defined by

$$\begin{aligned}&\delta_R((q_I, \mathcal{Q}_O), (a_I, a_O)) = \\ &(\delta_I(q_I, a_I), \\ &(\{\delta_O(q_O, a_O) \mid q_O \in \mathcal{Q}_O\} \cup \{q_O^0 \mid q_I \in F_I\}) \setminus F_O) \ .\end{aligned}$$

The transition function $\delta_R((q_I, \mathcal{Q}_O), (a_I, a_O))$ calculates the subsequent state $(p_I, \mathcal{P}_O) \in Q_I \times 2^{Q_O}$ based on the transitions of the automata A_I and A_O. The first part p_I simply reflects the state of the input automaton A_I having processed the input a_I. In contrast to the input automaton, the output can be in different states simultaneously. For each state $q_O \in \mathcal{Q}_O$ the subsequent state of the output automaton A_O is calculated separately. Besides these states, the state q_O^0 is included in $\mathcal{P}_O$ whenever the input automaton reaches an accepting state. Informally, this means that the input pattern is matched and the output automaton has to be started again. All accepting states of A_O have to be removed from this set, as the guarantee of the output has been fulfilled in these states. A finite violation of the required I/O behavior occurs, iff at least one of the accumulated output transitions results in the sink state s_O, *i. e.,* the corresponding output constraint can not be fulfilled anymore. Accordingly, in this case the resulting state $(p_i, \mathcal{P}_O)$ is an accepting state of A_R since $s_O \in \mathcal{P}_O$. ■

Using this automaton we can again construct the violations for the segments and the entire specification using the union of the row automata (assumed the row input languages in a segment are disjunct[5]). Checking for inputs without valid outputs can then easily performed by iteratively finding "bad" states, which will lead to accepting or other "bad" states for certain inputs regardless of the output produced by the system.This can be easily performed in polynomial time, leading to the following corollary.

Corollary 2. *Checking the consistency of a stream-based I/O table only describing safety properties can be performed in doubly exponential time, if the involved languages are given as regular expressions.*

The doubly exponential complexity is due to the conversion of the languages to DFAs followed by the power set construction. However, these operations are performed on automata which are rather small and in practice the exponential case seldom occurs. Additionally, all operations involved can be implemented easily compared to the corresponding operations for ω-regular languages.

VI. Practical Application

This section introduces prototypical tool support for our technique and summarizes a small industrial case study. The experience from both is discusses at the end.

A. Tool Support

As a basis for a small case study presented later, as well as for getting practical results on the efficiency of the analysis algorithms, we implemented an editor for our specification technique as an extension of the modelling tool AF/STEM[6]. AF/STEM is a specification environment for spatio-temporal systems in the domain of automation engineering. Systems are described by means of components with the syntactic interface given by ports and the semantics usually given by an automaton. We implemented stream-based I/O tables as an alternative way of specifying a component and provided editor support as well as an implementation of checks for input completeness and consistency.

A naive implementation based on the automaton construction presented in Sec. V-C is not feasible, as the alphabet of the automata involved usually is rather large. To circumvent this we used BDDs for encoding the transition labels, thus the number of transitions is bounded by the square of the number of states, which turned out to be of reasonable size for real world examples. An alternative would have been the transformation of our tables to a suitable solver (*e. g.,* MONA [23]), but our implementation has the advantage to allow direct access to the constructed automaton, which can be used for further steps, such as generation of test cases or controller code.

As there are no reference models to be used and our implementation is still rather simple with no optimizations or heuristics, we did not perform major benchmarks. However, even the largest tables of our case study, which is detailed in the next section, could be checked for input completeness and consistency within only a couple of seconds.

B. Case Study

To learn about the applicability of our technique, we formalized a considerable part of a textual requirements specification provided by Siemens, which describes a machine from the automation domain (already mentioned in Sec. I). The formalized part describes the controlling of different hardware components of a bottling plant, dealing with transport, decollating, and routing of bottles through the plant.

The resulting specification consists of 51 tables and 60 segments with between 4 and 10 rows and 3 to 7 columns each. Our simple approach turned out to be powerful enough to formalize all of the typical requirements of discrete event-based systems. Moreover, the case study showed that a large part of the requirements could be formalized by very simple patterns. As the relation between tables (51) and segments (60) indicates, it turned out to be more convenient to decompose a functionality into further sub-components instead of specifying complex relations by different segments. While segments are defined over the interface of the whole component, the advantage of the decomposition into sub-components is that the interfaces of the sub-components only comprise a subset of relevant channels. Therefore the tables stay considerably small.

In this case study, all requirements could be formalized with the presented method. However, for some requirements a purely sequence oriented approach was not applicable. Instead we introduced states as described in the next section. An example for the use of states is the decollating of bottles for a certain processing step (*e. g.,* filling, draining, topping). A first sensor indicates if there is a bottle ready to be decollated. To decollate a bottle the previous bottle has to be processed and must have passed a second sensor. The processing of the bottle takes an indefinite amount of time. While we were unable to formalize the fact that no bottle can be decollated while the respective sector of the plant is occupied using only sequences, we easily succeed by introducing a state capturing this information.

On the other hand – as expected – counting could be realized very easily by means of our tables. In the case study, we used this to model the change of the operation mode of the plant from automatic to manual or vice versa. In this case, some components have to be blocked for a certain amount of time (*e. g.,* five time intervals) to assure that the running operations are finished properly.

Our overall impression was, that despite its simplicity, our approach is suitable to describe not only academic examples but also real discrete event-based systems. Besides, the formalization showed that the underlying textual specification was incomplete, vague and ambiguous. The simple structure and syntax of the tables helped, when discussing details of the

[5]We may assume that the languages $I_1, \ldots, I_r$ are pairwise disjunct; otherwise we could transform the segment into the segment $\hat{S} = ((\hat{I}_1, \ldots, \hat{I}_r), O, r)$ with $\hat{I}_j = I_j \setminus \bigcup_{k=1}^{j-1} I_k$ which is semantically equivalent and has the desired property.

[6]http://af3.in.tum.de/index.php/AF/STEM

specification with engineers, who could understand the tables after only a short introduction.

C. Discussion

This section summarizes some findings and consequences from our notation and experiments.

Dealing with States: As argued before a major advantage over other state- or table-based specifications is that no state space has to be explicitly constructed beforehand. On the other hand portions of the requirements specification might already mention states explicitly in their description or can be simplified considerably when introducing states. We can easily integrate states in our tables by introducing an additional input and output channel which carries the state information. These channels are not externally available, but can be thought of as being connected internally by a loop. Thereby, our specification technique offers not only access to the actual state but to the whole state history. We can model state variables using the same pattern.

Language Extensions: Our experiments showed that syntactic sugar for expressing dependencies between entries in different cells of a table row are quite useful. As long as we only extend the concrete syntax, the presented results stay valid. Care has to be taken to not lose simplicity by overloading notation. An enhancement that has proved useful is to optionally require the matched input of a row to be of same length for all columns. This can express that certain events have to occur at input channels at the same (previous) time. To include it in our tables, only a minor modification in the construction of the row automata is required. Furthermore, operations on signals (*e. g.,* output is sum of inputs) are very complicated to describe with our approach. To overcome this problem, we plan to expand the used regular expressions by variables globally bound per row.

Size of the Type's Carries Set: While our specification technique can deal with types having large or even infinite carrier sets (such as integer numbers), our analysis procedures share the problems of model checkers, namely the exponential dependency on the input alphabet's size. This was not a problem in our case study, as all components dealt with simple signals only. However, when dealing with measurements, as for example speed, our analysis algorithms will not scale. One approach would be the use of abstraction or quantization, *i. e.,* instead of dealing with a speed signal we would only differentiate between low, normal, and high velocity. One could also try to automatically find a suitable abstraction from a table, but we did not yet investigate this.

VII. Conclusion

In this paper, we presented a simple technique for the specification of the behavior of reactive control systems. The technique is based on tables of regular expressions which makes the specifications easier to use for non-mathematicians. The very small set of primitives and the usage of well-known regular expressions support fast learning and understanding of these tables. It turns out that the pattern-based description often is closely related to textual requirement specifications. Consequently, our approach provides a smoother transition to formal methods. At the same time the formal foundation of those tables allows for an extensive analysis of the specified requirements which can be performed automatically as presented in this paper.

So far we only showed how to formalize functional requirements of a single component. If systems become larger and more complex, a decomposition of the system's functionality into a hierarchy of sub-functionalities is inevitable. The idea is to specify the different sub-functionalities independently by different I/O-tables and combine them afterwards. As described in [24], *services* are a suitable means to structure the functionality during the requirements engineering phase. Each service models a separate usage functionality of a system observable at its boundaries. Services may overlap, *i. e.,* be defined over the same input or output ports. As a consequence, they might intentionally or unintentionally influence each other (feature interaction). We are currently working on theoretical and methodological extensions of our technique for dealing with the questions arising in the context of service-based specifications.

References

[1] ITU-T, "Recommendation Z.120. Message Sequence Charts," International Telecommunication Union, Genève, Tech. Rep. Z-120, 2000.

[2] Object Management Group, "UML 2 superstructure specification," pp. 403–454, 2004. [Online]. Available: http://www.uml.org/

[3] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comp. Prog.*, vol. 8, no. 3, pp. 231–274, 1987.

[4] M. von der Beeck, "A comparison of statecharts variants," in *Proc. of FTRTFT*. Springer, 1994.

[5] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[7] C. Denger, D. M. Berry, and E. Kamsties, "Higher quality requirements specifications through natural language patterns," in *Proc. of SWSTE'03*. IEEE, 2003.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. of ICSE'99*. IEEE, 1999.

[9] A. Bauer, M. Leucker, and J. Streit, "SALT—structured assertion language for temporal logic," in *Proc. of ICFEM'06*. Springer, 2006.

[10] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," Jerusalem, Israel, Tech. Rep., 1998.

[11] M. Brill, W. Damm, J. Klose, B. Westphal, and H. Wittke, "Live sequence charts: An introduction to lines, arrows, and strange boxes in the context of formal verification," in *SoftSpez Final Report*, 2004.

[12] B. Sengupta and R. Cleaveland, "Triggered message sequence charts," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 587–607, 2006.

[13] C. L. Heitmeyer, J. Kirby, B. G. Labaw, and R. Bharadwaj, "SCR*: A toolset for specifying and analyzing software requirements," in *Proc. of CAV'98*. Springer, 1998.

[14] P.-J. Courtois and D. L. Parnas, "Documentation for safety critical software," in *Proc. of ICSE'93*. IEEE, 1993.

[15] C. L. Heitmeyer, "Applying practical formal methods to the specification and analysis of security properties," in *MMM-ACNS*, 2001.

[16] R. Janicki and R. Khédri, "On a formal semantics of tabular expressions," *Sci. Comput. Program.*, vol. 39, no. 2-3, pp. 189–213, 2001.

[17] C. L. Heitmeyer, M. Archer, R. Bharadwaj, and R. D. Jeffords, "Tools for constructing requirements specifications: the scr toolset at the age of nine," *Comput. Syst. Sci. Eng.*, vol. 20, no. 1, 2005.

[18] D. K. Peters, M. Lawford, and B. T. y Widemann, “An IDE for software development using tabular expressions,” in *Proc. of CASCON’07*. IBM, 2007.

[19] D. L. Parnas, “Tabular representation of relations,” Telec. Research Institute of Ontario, CRL Report 260, 1992.

[20] M. Herrmannsdörfer, S. Konrad, and B. Berenbach, “Tabular notations for state machine-based specifications,” *Crosstalk*, vol. 21, no. 3, pp. 18–23, 2008.

[21] J. E. Hopcroft and J. D. Ullman, *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[22] W. Thomas, “Automata on infinite objects,” in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, 1990, pp. 133–192.

[23] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm, “Mona: Monadic second-order logic in practice,” in *Proc. of TACAS’95*. Springer, 1995.

[24] A. Harhurin and J. Hartmann, “Towards Consistent Specifications of Product Families,” in *Proc. of FM’08*. Springer, 2008.