Formal Semantics for Tabular Expressions and Software Cost Reduction Method

FORMAL SEMANTICS FOR TABULAR EXPRESSIONS AND SOFTWARE COST REDUCTION METHOD

By

IMENE BOURGUIBA, B.Sc., M.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2011) McMaster University

(Computer Science) Hamilton, Ontario

TITLE: Formal Semantics for Tabular Expressions and Software Cost Reduction Method

AUTHOR: Imene Bourguiba, B.Sc., M.Sc.

SUPERVISOR: Dr. Ryszard janicki

NUMBER OF PAGES: xvi, 156

Acknowledgements

I am thankful to my supervisor Dr. Ryszard Janicki, for his support and guidance.

Deepest gratitude are also due to the members of my supervisory committee, Dr. Farmer and Dr. Sekerinski.

I would like to especially thank Dr. Moa for many fruitful and stimulating discussions.

I wish to express my love and gratitude to my beloved mother and my husband, for their understanding and endless love, through the duration of my studies.

I am heartily thankful to the flowers of my life Eya and Alaa. Their laughs and smiles brought joy to my life, and gave me strength during my PhD journey.

I would like to thank all people who have helped and inspired me during my doctoral studies especially my brother, my sisters, and my friends.

Lastly, and most importantly, I dedicate my dissertation to the memory of my beloved father.

Abstract

Unambiguous and precise software specification can not be achieved without some use of formal notation. Table-based specification techniques are both readable and convenient. They allow the representation of systems specifications in a very compact and yet precise manner. They scale to software systems, and they may be easily used even by people unfamiliar with the application domain. Additionally, the use of table-based notations makes it relatively easy to check for such properties as consistency and completeness. Among the table-based specification techniques discussed in the literature, the most popular are the Software Cost Reduction (SCR) method and tabular expressions. Both of these techniques are successfully used in practice to formally specify software requirements.

The Software Cost Reduction (SCR) method is a formal method for specifying the requirements of software systems that is based on tabular notation. SCR is used in a wide range of applications.

The second technique tabular expressions comprises a collection of cells, with each cell holding a single expression. The beauty of tabular expressions stems from both their visual structure and their concise representation of mathematical functions and relations. As a result, these expressions are suitable for use in every software engineering phase, from establishing requirements to completing final testing.

To successfully be used in practice, the specification techniques chosen should be supported by tools for creating, editing and transforming tables. Creating tools in the absence

of reasonable formal semantics often results in failure. Formal semantics are also needed to compose and decompose tables in a modular way.

Although SCR has been used in many projects and organizations to specify software requirements, perhaps surprisingly, its semantics are not well defined. Specifically, the symbols used in this method are ambiguous, especially those that serve to denote SCR events. Further, the SCR method does not include either table composition or decomposition.

The tabular expressions technique was also lacking, though in different ways than SCR. A literature review revealed that the techniques applied to address the challenges inherent in tabular expressions composition have their own limitations.

The aim of this research, then, was to improve the semantics of both the SCR and the tabular expressions specification methods. To this end, SCR tables were converted into tabular expressions, as they have a rather precise semantics. Additionally, a new way to model the SCR events with first order logic is presented. Finally, a simpler way to define SCR events with propositional logic is proposed.

By improving the semantics of the two specification methods, numerous advantages are realized. These include increasing the readability of tables, and eliminating previously ambiguous symbols.

Moreover, improving the semantics enabled certain tasks to be carried out more easily, such as facilitating the verification and validation process and improving the toolset supporting the SCR method. In moving towards a richer semantics, this research allowed for the introduction of algebra for tabular expressions, as well as operators for tables composition and decomposition. Further, the research revealed the inherent power of tabular expressions. This was accomplished by composing a regression to demonstrate where tabular expressions get their power in specifying functions, relations and programs. An application of tabular expressions for three dimensions and higher is successfully presented. Next, a language and a structure for tabular expressions is proposed. Then, it is shown how tabular

expressions could be represented by a lattice and by a vector space, respectively. Finally, the discussion considers the ways in which such an enhanced tabular expressions application could also be applied to other fields, such as software engineering and computer science.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal notations are needed in order to have precise and unambiguous specifications [40]. Table-based specification techniques allow one to represent systems specifications in a compact and precise manner. Tabular or table-based notations utilize dimensional space, and are especially useful when many cases have to be considered or when functions are built from elements of different types [28]. They scale to software systems and make the checking of important properties, such as consistency and completeness, quite natural and relatively easy [18]. In this thesis, we start by presenting table-based specifications techniques. The most important techniques that we found in the literature are tabular expressions, Software Cost Reduction (SCR), Requirements State Machine Language (RSML), function tables, and decision tables. These techniques are used to specify and analyse software systems. They rely on a tabular notation, which is very helpful for increasing their readability. They have a syntax and semantics of their own. Moreover, most of them encompass the verification and validation process. We close the thesis by discussing and comparing more detailed tabular expressions and SCR. Both are successfully used in practice, especially to formally specify software requirements.

1.1 Motivation

In order to be used in practice, techniques should be supported by tools for creating, editing, and transforming tables, etc. Creating tools without a reasonable formal semantics may and often will lead to their failure. Our research objective is to improve the syntax and semantics of both SCR and tabular expressions. These two methods were chosen because they are successfully used in practice.

SCR semantics is not well defined. In fact, the symbols adopted by the method are ambiguous. Therefore, we transformed SCR tables into tabular expressions, since the latter have a precise semantics. We present a new way to model the SCR events with first order logic. We also depict another way to define SCR events with propositional logic.

For tabular expressions, as a step towards a richer semantics, we propose an algebra for tabular expressions. In our algebraic model, we introduce new operators that allow us to compose and decompose tabular expressions. The operators we came up with are classified into unary operators, inner operators, Kronecker operators, and outer operators. We also propose a refinement ordering relation on tabular expressions, and we depict an application of a tabular expression for three dimensions and higher. To the best of our knowledge this is the first time such an application has been proposed. Next, we present a language and a structure for tabular expressions. Finally, we show how tabular expressions can be represented by a lattice and by a vector space respectively.

By reaching our goals, we can build quality professional tools based on the new semantics. The two methods can be applied not only at the requirements level, but also at all every phase of the software development process.

1.2 Outline

In Chapter 2, we present five table-based specification techniques.

Among the presented techniques, we discuss in details the Software Cost Reduction method and tabular expressions in Chapter 3. Also, we present the semantics of the two methods, and we show how to convert SCR tables into tabular expressions, and how to improve SCR semantics.

Chapter 4 presents the relational composition problem. We survey the literature regarding the relational/table composition problem, and we present examples to illustrate that.

Chapter 5 discusses the operators we developed, which are applied to tabular expressions, the partial order and the refinement ordering relation specified on tabular expressions, as well as the algebra of tabular expressions.

In Chapter 6, we present the application of tabular expressions for three dimensions and higher. Also, we depict a language and a structure for tabular expressions. Then, we show how tabular expressions can be represented by a lattice and by a vector space respectively.

Finally in Chapter 7, we conclude and give directions for future work.

In Appendix A, we show how to extend Janicki's work to compose tables instead of composing cells of one table. We give examples of composition of normal tables, inverted tables, vector tables, and decision tables.

In Appendix B, we depict how function tables fit in the general framework of tabular expressions.

Chapter 2

Table-based specification techniques

2.1 Introduction

Unambiguous and precise software specification can hardly be achieved without some use of formal notation. Table-based specification techniques are readable and convenient to specify systems requirements. They permit one to depict systems specifications in a compact and precise manner. Besides, they make the checking of important properties such as consistency and completeness quite natural and relatively easy. For requirements consistency, we have to check that there are no conflicting requirements or unwanted nondeterminism. For requirements completeness, we have to ensure that the set of requirements is complete, which means that there is a response for every possible input [13].

In this chapter, we present and analyse five table-based specifications techniques which are tabular expressions, Software Cost Reduction (SCR), Requirements State Machine Language (RSML), function tables, and decision tables. A similar discussion on these techniques could be found at [5]. In Sections 2.2 to 2.6, we briefly describe each technique. In Section 2.7, we discuss the verification and validation process for the presented techniques. In Section 2.8, we depict the merits and disadvantages of the discussed techniques.

2.2 Tabular expressions

Tabular expressions are a generalisation of two dimensional tables as classical decision tables and state transition tables that date back to early years in computer science [23]. Parnas pioneered the use of tabular expressions to document software requirements [21]. In the late 1970s, these tables were practically used in an ad hoc manner. The practical experience showed that these tables are really suitable to present functions and relations. Early versions were adopted by the U. S. Naval Research Laboratory in 1977 to write a software requirements document for the Onboard Flight Program used in the U.S Navy's carrier based attack aircraft, the A-7E [21]. Then in 1992, Alspaugh et al. [2] introduced the software requirements specifications for the A-7E Aircraft representing the evolution of the SCR project since 1977. Tables were chosen to ensure conciseness and completeness that were not offered by textual format. In [48], the A-7 requirements model was re-examined for real time systems application, and tables were improved. Tables were also adopted for the software for the Darlington Nuclear Generating Station Shutdown System [10, 53], and by the Software Quality Research Laboratory at McMaster University [31, 32, 45, 41, 54].

In [38], Parnas, presented for the first time a formal syntax and semantics for tabular expressions. He proposed ten classes of tables. His classification was based on the types of applications. More general application independent formal semantics was proposed by R. Janicki in [25] and then refined in [26, 28]. Each class proposed by D.L. Parnas in [38] could be seen as a special case of the generic model proposed by R. Janicki in [25, 26, 28]. In principle a table is represented by a collection of cells where each cell holds an expression. In [39], expressions are defined as an indexed set of mathematical variables, and a set of constants. Tabular expressions are very convenient and very readable especially for long and complex formulae. We will illustrate them by showing how they can be used to formally specify scenarios.

In [7, 9], a formal scenario is defined as a 3-tuple (T, R_e, R_s). This tuple represents a relational model of an informal scenario. The set T is the scenario's space or domain, while R_e and R_s are two disjoint relations representing the relation of the environment, and the relation of the system respectively. The relation of the scenario is given by the relation $R_e \cup R_s$. The relation R_e is a description of the behavior of the environment to the system according to the scenario. The relation R_s is a specification of the system as reported by the scenarios. Tabular expressions can easily be adopted to represent the relations of such formal scenarios. To illustrate this, let us consider the following informal scenario.

Open Deposit Account : If the person is already holding an *account* (Account Holder), then the response is "This person is already an account holder". If a person is not already an account holder and if the amount offered to open an *account* is greater than *MinAmount*, then the *account* is opened and a debit card is assigned to it. The opening *balance* is the amount offered minus the opening fee. The *bank's funds* position is increased by the amount offered to open an *account* and the response is "Account opened". Otherwise, if the amount offered is less than *MinAmount*, then the response is "Overdrawn balances are not allowed". From this informal scenario, the formal scenario given by the space, the relation of the environment and the relation of the system are represented by Table 2.1, Table 2.2 and Table 2.3 respectively (λ represents the empty sequence). The 'prime' notation proposed in [38] is used when writing appropriate predicates. It denotes the new value of the variable. For example the programming sentence `if x<0 then x=x+1` translated into 'prime' notation looks like $x < 0 \wedge x' = x + 1$. The variables are state space component. For conciseness of the notation, only the changed variables are presented, and the other variables remain unchanged. The relation of the scenario is presented in Table 2.4. Tables 2.1, 2.2, 2.3 and 2.4 are practically self explained. For more details about scenario formalisation and how in practice to calculate $R_e \cup R_s$, we refer the interested reader to [7, 9].

Tabular expressions are well suited to the design level [51]. In [33], the result of the

Variable Name	Variable Domain
Account	$Names$
Holder	$Names$
Output	$\{\text{exist}, \text{overdrawn}, \text{account}-\text{opened}\}$
setOfAccountHolders	2^{Names} (set of all names)
Amount	$Real$
Balance	$[\text{Account} \rightarrow Real]$
BankFund	$Real$
DebitCard	$[\text{Account} \rightarrow \{\text{assigned}, \text{not}-\text{assigned}\}]$
OpenAccount	$[\text{Holder} \rightarrow \text{Account}]$

Table 2.1: Scenario's space T

	Amount < 0	$0 \leq$ Amount $<$ Min	Amount $\geq$ Min
Holder $= \lambda$	Holder'$\in Names$ $\wedge$ Amount' $\in Real$	false	false
other	false	false	false

Table 2.2: The relation of the environment R_e

	Amount < 0	$0 \leq$ Amount $<$ Min	Amount $\geq$ Min
Holder $\neq \lambda$ $\wedge$ Holder $\in$ setOfAccountHolders	false	Output'=exist $\wedge$ Holder' $= \lambda$ $\wedge$ Amount'< 0	Output'=exist $\wedge$ Holder' $= \lambda$ $\wedge$ Amount'< 0
Holder $\neq \lambda$ $\wedge$ Holder $\notin$ setOfAccountHolders	false	Output'=Overdrawn $\wedge$ Holder'=λ $\wedge$ Amount'< 0	Output'=account-opened $\wedge$ setOfAccountHolder'= setOfAccountHolders$\cup${Holder} $\wedge$ BankFund'=BankFund+Amount $\wedge$ Account'=OpenAccount(Holder) $\wedge$ DebitCard(Account')=assigned $\wedge$ Balance(Account')=Amount-Fees $\wedge$ Holder' $= \lambda$ $\wedge$ Amount'< 0
other	false	false	false

Table 2.3: The relation of the system R_s

	Amount < 0	$0 \leq$ Amount $<$ Min	Amount $\geq$ Min
Holder = λ	Holder' $\in A^+$ $\wedge$ Amount' $\in R^+$	false	false
Holder $\neq \lambda$ $\wedge$ Holder $\in$ setOfAccountHolders	false	Output'=exist $\wedge$ Holder' $= \lambda$ $\wedge$ Amount'< 0	Output'=exist $\wedge$ Holder' $= \lambda$ $\wedge$ Amount'< 0
Holder $\neq \lambda$ $\wedge$ Holder $\notin$ setOfAccountHolders	false	Output'=Overdrawn $\wedge$ Holder'=λ $\wedge$ Amount'< 0	Output'=account-opened $\wedge$ setOfAccountHolder'= setOfAccountHolders$\cup${Holder} $\wedge$ BankFund'=BankFund+Amount $\wedge$ Account'=OpenAccount(Holder) $\wedge$ DebitCard(Account')=assigned $\wedge$ Balance(Account')=Amount-Fees $\wedge$ Holder' $= \lambda$ $\wedge$ Amount'< 0
other	false	false	false

Table 2.4: The relation of the scenario

integration of all formal scenarios can be used to derive the functional architectural design of simple systems. Besides, when the requirements are documented with tabular expressions, it is easy to derive from them random testing, software integration test cases, and oracles. In [34], test cases were generated from the tabular expressions representing formal scenarios.

In [54], tabular expressions were adopted to model concurrent systems. A concurrent system is considered as a collection of processes and shared objects. Once the processes and the shared objects are determined, each process is represented by a Finite State Automaton (FSA). Afterwards, these machines are merged in order to come up with a global FSA which defines the concurrent system. Tabular expressions were adopted to depict the transition function of the global finite state automata. Adopting a tabular notation to represent the transition function reduces the size of the table considerably. Indeed, an automaton with m states and n events, has a transition function with m rows and n columns. For large values of m and n, and when combining different processes, the table size could grow exponentially. However, Y. Yang and R. Janicki showed that with tabular expressions, the number of rows and columns of the table representing the transition function decreased

considerably. Also their work can be extended to add priority [54].

At McMaster University, the Software Quality Research Laboratory (SQRL) developed a set of tools called Table Tool System (TTS) manipulating tabular expressions. Many modules have been implemented to allow some operations on tables such as creating, editing, analysing, interpreting, etc. The toolset is based on a set of modules that can be extended to add other modules and without being knowledgeable about the previous ones [41].

2.3 Software Cost Reduction

The Software Cost Reduction (SCR) was originally developed in U.S. Naval Research Lab to document the requirements for the A-7E aircraft [2], then it was successively improved by a team led by C.L. Heitmeyer [19]. It is one of the most popular formal method based on a tabular notation for specifying the requirements of software systems. The SCR requirements specification presents the system behaviour and environment. The environment involves the controlled variables (quantities that the system controls), and monitored variables (quantities that the system monitors). The environment generates a sequence of monitored events, and the system reacts to the monitoring events by changing their states. It is represented by a state machine $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, S_0 is the set of initial states ($S_0 \subseteq S$), E^m is the set of monitored events, and T is the transform function, which from a current state $s \in S$, and an event $e \in E^m$ returns the next state $s' \in S$. The SCR state machine model is a special case of Parnas' Four Variable Model (FVM) [37]. There are a couple of slightly different versions of the FVM. The one used here is developed by Parnas and Madey to specify system requirements [40]. It is an extension of the classical Two Variable Model (input and output) [37]. The FVM consists of four sets of variables and four relations (see Figure 2.1). M represents the variables monitored by the system. C depicts the variables controlled by the system. I outlines the input vari-

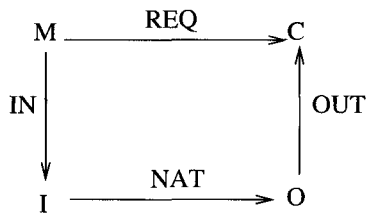


Figure 2.1: Four variable model

ables. O designates the output variables. The relation NAT defines the natural constraints required by the environment. The relation REQ presents the required system behaviour. The relation IN describes the behaviour of the input. It maps the monitored variables into input variables. The relation OUT describes the behaviour of the output. It maps the output variables into the controlled variables.

The SCR formal model uses only the relations NAT and REQ to define the system behaviour. In order to have a more concise specification, some constructs such as mode classes and terms were added to the SCR model. The values of mode classes are modes. Modes are classes of system states specifying the system behaviour. With SCR, each specification is organized into dictionaries and tables. The dictionaries represent static information such as variables names and types, whereas the tables depict the variables changes while responding to input events. In SCR, there are three kinds of tables to specify a system: condition tables, event tables, and mode transition tables. The tables discussed here describe a safety injections system and are borrowed from [15].

A condition table defines a variable according to a mode and a condition. A condition is a predicate defined on a system state. For example, Table 2.5 identifies the controlled variable SafetyInjection as a function of Pressure and the term Overriddern. For instance the first column of Table 2.5 indicates that if the Pressure is High or Permitted, or if the Pressure is TooLow and Overridden is True then SafetyInjection is Off.

An event table defines a variable according to a mode and an event. An event represented

Mode Class Pressure	**Conditions**	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
Safety Injection	Off	On

Table 2.5: Condition table for *Safety Injection*

Mode Class Pressure	**Events**	
High	False	@T(Inmode)
TooLow, Permitted	@T(Block=On) WHEN Reset=Off	@T(Inmode) OR @T(Reset=On)
Overridden	True	False

Table 2.6: Event table for *Overridden*

by @T(c) means that condition c changes from false to true. For example, @T(Block=On) when T(Reset=Off) means that the operator turns Block from Off to On when Reset is Off. The @T(Inmode) means that the system enters into the class of modes in that row. In Table 2.6, the mode Pressure is defined via the current mode, and the events defined on the variable WaterPress. In Table 2.6, cell (2,1) indicates that if the Pressure is TooLow or Permitted, and Block changes to On When Reset is Off, then Overidden changes to True. A mode transition table generates a destination mode from a mode and an event. In the first row of Table 2.7, we see that if the Pressure is TooLow and WaterPres is greater than or equal to Low then Pressure becomes Permitted.

SCR has a toolset SCR* which contains an editor, a dependency graph browser, a simulator, a consistency checker, and a verifier. The specification editor enables one to create and modify requirements specification. The dependency graph browser depicts the dependencies between the variables. The simulator allows the system to be executed. The

Old Mode	Event	New Mode
TooLow	@T(WaterPress $\geq$ Low)	Permitted
Permitted	@T(WaterPress $\geq$ Permit)	High
Permitted	@T(WaterPress $<$ Low)	TooLow
High	@T(WaterPress $<$ Permit)	Permitted

Table 2.7: Mode transition table for *Pressure*

consistency checker permits one to check the specification for properties such as correctness and completeness, while the verifier has the role to verify and analyse some properties of the specification.

2.4 Requirements State Machine Language

The Requirements State Machine Language (RSML) was developed by the Irvine Safety Research Group to specify the TCAS II (Traffic Alert and Collision Avoidance System), a complex aircraft collision avoidance system [13]. It is based on hierarchical state machines. It combines a tabular notation with a graphical representation based on Harel statecharts. Harel statecharts are an extension of state-transition diagrams. They deal with hierarchy, concurrency, and communication [11]. Since RSML is based on Harel statecharts, it supports the notion of superstates which are helpful in reducing the number of transitions. In fact, the transitions could be from and to the superstate instead of having them from and to sub-states. For instance in Figure 2.2, transition A ends at the superstate. Transition C exits the superstate, which means it exits both of the states inside it, states R and S in this case. Another important feature supported by RSML are the parallel states[1]. They contain more than a state machine separated by dashed lines. For instance in Figure 2.3,

[1]Called also "AND states", "orthogonal products", or "product states".

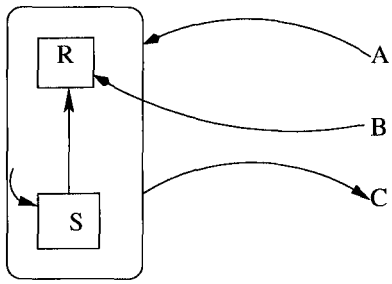


Figure 2.2: A superstate example

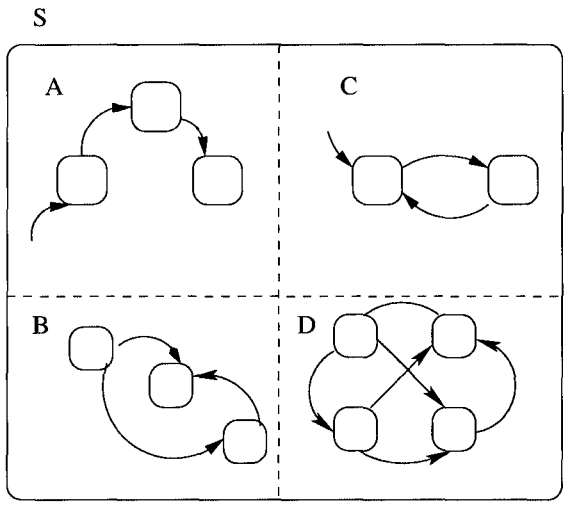


Figure 2.3: Parallel state

when the parallel state S is entered, all state machines A, B, C and D are entered as well. The states are exited whenever a transition exits from the parallel state. These parallel states decrease considerably the size of the specification. Transitions definitions in RSML contain five parts, which are the source and destination of the transition, its location, the triggering event, the output action which determines the events generated by the transitions and the guarding conditions. At the beginning, the guarding conditions of the transitions were written with predicate calculus as shown in Figure 2.4 (example borrowed from [13]). However, the TCAS external reviewers had difficulties to read such expressions. In ad-

$$
\begin{array}{c}
\text{True-Tau-Capped}_{f-362} \geq \text{Time-To-CPA} \wedge \\
(\text{Other-Capability}_{v-212} \neq \text{TCAS-TA/RA} \vee \\
(\text{Other-VRC}_{v-209} = \text{No-Intent} \wedge \text{Two-Of-Three}_{m-327})) \wedge \\
((\text{Down-Separation}_{f-337}(\text{low-firm}) \leq \text{Alt-Threshold} \wedge \\
\text{Up-Separation}_{f-362}(\text{low-firm}) \leq \text{Alt-Threshold}) \vee \\
(\text{Current-Vertical-Separation}_{f-332} > 150 \text{ ft} \wedge \\
((\text{Inhibit-Biased-Climb}_{f-339}(\text{low-firm}) > \text{Down-Separation}_{f-337}(\text{low-firm}) \wedge \\
\text{Own-Tracked-Alt}_{f-349} < \text{Other-Tracked-Alt}_{f-334}) \vee \\
(\text{Inhibit-Biased-Climb}_{f-339}(\text{low-firm}) \leq \text{Down-Separation}_{f-337}(\text{low-firm}) \wedge \\
\text{Own-Trarked-Alt}_{f-349} > \text{Other Tracked-Alt}_{f-334})))
\end{array}
$$

Figure 2.4: Transition condition written in predicate calculus

$\text{Other-Capability}_{v-212} = \text{TCAS-TA/RA}$	.	.	.	F	F	F
$\text{Other-VRC}_{v-209} = \text{No-Intent}$	T	T	T	.	.	.
$\text{Two-Of-Three}_{m-327} =$	T	T	T	.	.	.
$\text{True-Tau-Capped}_{f-362} < \text{Time-To-CPA}$	F	F	F	F	F	F
$\text{Down-Separation}_{f-337}(\text{low-firm}) \leq \text{Alt-Threshold}$	T	.	.	T	.	.
$\text{Up-Separation}_{f-362}(\text{low-firm}) \leq \text{Alt-Threshold})$	T	.	.	T	.	.
$\text{Inhibit-Biased-Climb}_{f-339}(\text{low-firm}) > \text{Down Separation}_{f-337}(\text{low-firm})$	.	T	F	.	T	F
$\text{Own-Tracked-Alt}_{f-349} < \text{Other-Tracked-Alt}_{f-334}$	.	T	.	.	T	.
$\text{Own-Tracked-Alt}_{f-349} > \text{Other-Tracked-Alt}_{f-334}$	.	.	T	.	.	T
$\text{Current-Vertical-Separation}_{f-332} > 150$ ft	.	T	T	.	T	T

Table 2.8: The AND/OR table

dition, such notation did not scale for complex expressions [12]. Hence, AND/OR tables were adopted to specify the guarding conditions, and to increase readability. They contain Boolean formulas expressed in Disjunctive Normal Form (DNF). Table 2.8 is an example of an AND/OR table. The dot means "do not care". The leftmost column of the table presents the predicates. A column is true if all its elements are true. If a column is true, then all the table is evaluated to true as well.

NIMBUS is a requirements engineering environment where the RSML specification could be executed. It allows one to evaluate an RSML specification dynamically. Also, it

enables engineers to execute the RSML specification early in the project. Once the specification is refined, engineers could integrate some simulations or details on the embedding or physical environment. So, with NIMBUS, the RSML specification is executed with the interaction with other models such as user input, software simulations of the components, models of the environment, etc.

2.5 Function tables

Function tables were successfully used to define requirements for the Darlington Nuclear Power Plant Shutdown System [36, 10, 51]. In principle they use functions to describe system's functionality for safety critical software systems. Technically, the function tables are a special kind of tabular expressions [51], probably the most important kind of tabular expressions. Function tables were constantly improved to satisfy developers and requirements engineers. For instance, previous cumbersome symbols such as $\#, *$ used as delimiters were removed. Also, other changes such as modifying tables format, helped function tables to become more readable. There are four kinds of function tables: vertical condition tables, horizontal condition tables, structured decision tables, and state transition tables. The following tables are borrowed from [36].

In vertical condition tables, the left bottom cell indicates the name of the function. For example, in Table 2.9, the name of the function is: f-trip. The other columns indicate the value of the function when the condition of the respective columns is true. So, in this example, the function f-trip is interpreted as:

f-trip = e-tripped if ((m-level >level-limit) AND (m-enable = e-enabled))

f-trip = e-not-tripped if NOT ((m-level >level-limit) OR NOT (m-enable = e-enabled)).

In horizontal condition tables, the right cells on the top indicate the function names. The other rows indicate the conditions and the function values when the respectives conditions

VCT: trip

	(m-level >level-limit) AND (m-enable = e-enabled)	NOT(m-level > k-level-limit) OR NOT (m-enable = e-enabled)
f-trip	e-tripped	e-not-tripped

Table 2.9: Vertical condition table

HCT: foo-fee

	Result	
Conditions	f-foo	f-fee
m-Trip[1] = 1 AND m-Trip[1] $\neq$1	e-tripped	e-not-tripped
m-Trip[1] $\neq$1 AND m-Trip[2] = 1	e-tripped	e-not-tripped
m-Trip[1] $\neq$1 AND m-Trip[2] $\neq$ 1	e-not-tripped	e-not-tripped
m-Trip[1] =1 AND m-Trip[2] = 1	e-tripped	e-tripped

Table 2.10: Horizontal condition table

are true. From Table 2.10, we can see that the function f-foo is defined as follows:

f-foo = e-tripped if ((m-Trip[1] = 1) AND (m-Trip[2] $\neq$ 1)),

f-foo = e-tripped if ((m-Trip[1] $\neq$ 1) AND (m-Trip[2] = 1)),

f-foo = e-not-tripped if ((m-Trip[1] $\neq$ 1) AND (m-Trip[2] $\neq$ 1)),

f-foo = e-tripped if ((m-Trip[1] = 1) AND (m-Trip[2] = 1)),

and

f-fee = e-not-tripped if ((m-Trip[1] = 1) AND (m-Trip[2] $\neq$ 1)),

f-fee = e-not-tripped if ((m-Trip[1] $\neq$ 1) AND (m-Trip[2] = 1)),

f-fee = e-not-tripped if ((m-Trip[1] $\neq$ 1) AND (m-Trip[2] $\neq$ 1)),

Condition Macros:
w-trip-rng[m-ai, f-sp]
hitrp : m-ai $\geq$= f-sp
ddbnd : (m-ai < f-sp) AND (m-ai $\geq$ (f-sp - k-db))
notrp : m-ai < (f-sp - k-db)
SDT: trip

Condition Statements	1	2	3	4
w-trip-mg[m-ai, f-sp1]	hitrp	ddbnd	ddbnd	notrp
f-trip$_{-1}$ = e-tripped	-	T	F	-
Action Statements				
f-trip = e-tripped	X	X		
f-trip = e-not-tripped			X	X

Table 2.11: Structured decision table

f-fee = e-tripped if ((m-Trip[1] = 1) AND (m-Trip[2] = 1)).

Structured decision tables have conditions states, action states, and rules. Also, they may have conditions macros that come with the table to help shortening cells content, so that the table will not be cumbersome.

From Table 2.11, the function f-trip is interpreted as:

f-trip = e-tripped if (m-ai $\geq$ f-sp),

f-trip = e-tripped if ((m-ai < f-sp) AND

((m-ai$\geq$ (f-sp - k-db)) AND

(f-trip-1 = e-tripped)),

f-trip = e-not-tripped if ((m-ai < f-sp) AND

((m-ai $\geq$ (f-sp - k-db)) AND NOT

(f-trip-1 = e-tripped)),

f-trip = e-not-tripped if (m-ai < (f-sp - k-db)).

State transition tables represent next state functions. The top row contains the transition conditions enabling the states change. The leftmost column designates the reachable states. For instance, from Table 2.12, we see that when the system is in the state "e-time" and the

STT: f-digitalwatch

Transition Condition / **Previous State** ↓	m-select = e-pressed	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-pressed)	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-unpressed)	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-unpressed)	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-unpressed)
e-time	e-in-time	e-time	e-time	e-time	e-time
e-in-time	e-in-time	e-in-time	e-in-time	e-in-time	e-zero
e-zero	e-stopwatch	e-zero	e-running1	e-running2	e-running2
e-running1	e-stopwatch	e-running1	e-running1	e-running2	e-running2
e-running2	e-stopwatch	e-stopped1	e-stopped1	e-running2	e-running2
e-stopped1	e-stopwatch	e-stopped1	e-stopped1	e-stopped	e-stopped2
e-stopped2	e-stopwatch	e-stopped2	e-running1	e-stopped	e-stopped2
e-stopped	e-stopwatch	e-stopped	e-stopped	e-stopped	e-zero
e-stopwatch	e-stopwatch	e-time	e-time	e-time	e-time

Table 2.12: State transition table

condition "m-select = e-pressed" is TRUE, then the system will be in the "e-in-time" state.

It is very advantageous to adopt function tables. For instance, the safe state is easily identified. By placing it at the rightmost column, it is easily identified by developers and designers [53].

In [52], the developed tools support most of the development life-cycle phases. The requirements tool allows to check for consistency and completeness. Both the requirements tool and the design tool enable the generation of templates that are translated into PVS. At the coding phase, function tables were converted into FORTRAN code, and this conversion was very helpful for the verification [53]. Also, a simulation tool providing a platform for testing was developed.

2.6 Decision tables

Decision tables have been used for many years in software engineering [23]. There are many kinds of decision tables. In [22], Hoover and Chen adopted decision tables to define

((Flightphase = climb) AND (AC-Alt > 400) AND
(AC-Alt < Acc-Alt) AND (NOT Alt-Capt-Hold))
OR
((Flightphase = climb) AND (AC-Alt > 400) AND (AC-Alt < Acc-Alt) AND Alt-Capt-Hold
AND (Alt-Target > prev-Alt-Target))

Figure 2.5: The engagement criterion of Takeoff

Operational Procedure		Takeoff		Climb		Climb-Int-Level	Cruise
Input Variables	States						
Flightphase	climb cruise	climb	climb	climb	climb	climb	cruise
AC-ALt> 400	True False	True	True	*	*	*	*
Compare (AC-Alt, Acc-Alt)	LT EQ GT	LT	LT	EQ GT	EQ GT	*	GT
Alt-Capt-Hold	True False	False	True	False	True	True	True
Compare (Alt–Trgt, prev-Alt-Tgt	LT EQ GT	*	GT	*	GT	*	EQ

Table 2.13: Decision table

functions and relations. The semantics of decision tables is based on propositional logic. The tables contain only variable names and values. No logical symbols are used since the logical connectives are expressed by the table structure itself [22]. They specify different actions that the system has to consider. The actions are called "operational procedures". In a decision table, there are two parts. The top part depicts the conditions of the operational procedures called "engagement criterion". The bottom part outlines the behaviors of the operational procedures. For instance, in Table 2.13 (example borrowed from [22]), the engagement criterion of Takeoff is shown in Figure 2.5.

The kind of decision tables previously discussed are called simple decision tables. Partioned decision tables represent the same information in a more compact way. The rows and columns (separated by double lines) represent macro-rows and macro-columns where

Operational Procedure		Takeoff		Climb		Climb-Int-Level	Cruise
Input Variables	States						
Flightphase	climb cruise	climb		climb		climb	cruise
AC-ALt> 400	True False	True		*		*	*
Compare (AC-Alt, Acc-Alt)	LT EQ GT	LT		LT EQ		*	GT
Alt-Capt-Hold	True False	False	True	False	True	True	True
Compare (Alt–Trgt, prev-Alt-Tgt	LT EQ GT	*	GT	*	GT	*	EQ

Table 2.14: A Partitioned decision table

each cell is a decision table (see Table 2.14). They have the same semantics as the simple decision tables. To check the consistency and completeness of decision tables, Hoover and Chen introduced a tool named"Tablewise". It identifies the flaws causing the inconsistency and incompleteness of decision tables. Tablewise generates Ada code and English documentation from the functions presented by decision tables.

2.7 Verification and validation

The task of verification and validation of software products is very substantial, and it has been an important area of research for more than three decades. This claim is still true today, and many techniques are used for the verification and validation task ranging from informal to formal. At the requirements level, once the problem is understood, the task of verification and validation should be accomplished. At the verification stage one should ask: "Am I building the product right", while at the validation stage the question is: "Am I building the right product" should be asked [4]. For requirements verification, we check some properties such the consistency, completeness and unambiguity of the requirements [4]. For requirements validation, we seek whether there is a mismatch between

users' intentions and the requirements documents.

With their formal structure, tabular expressions are very convenient to check for consistency and completeness [28, 51]. Indeed, the disjointness of the headers of the table ensures the consistency of the specification. Then, the complete input domain coverage ensures that we have specified responses to every input combination. The domain coverage condition also can be easily verified. The exact verification formulae depends on the type of tabular expressions, but it is rather straightforward in each case [28, 51]. Tabular expressions are very helpful to verify partial completeness with the domain coverage theorem. In [9], tabular expressions were used to represent the relational scenarios describing the interactions between the system and its environment. It was proved that they simplify the task of verification and validation. In fact, with tabular expressions it is easy to integrate the scenarios when they are consistent, or to find the source of inconsistency otherwise. The inconsistency between scenarios is detected during their integration. When the scenarios are inconsistent, the client is asked for some clarifications about the inconsistency detected. The completeness verification could be done on individual scenarios, or after integrating the requirements.

The validation consists in checking the specification conformity to some of the user's expectations. In [7, 9], the validation is performed at different levels. For instance, completeness tests are performed on the integrated scenarios to ensure that the behaviour of the system is complete. There are also other tests that are accomplished, such as ensuring that there is no deadlock between the environment actions and system reactions.

In SCR, once the requirements are specified, properties such as consistency, completeness and correctness are checked [19]. For consistency, the disjointness property must be satisfied. It guarantees that in each row, the conjunction of any two cells is false. For completeness, the coverage property of condition tables must be satisfied. The disjunction of the conditions of each row must be true.

To ensure that the specification captures the intended behavior, the user runs a simulator, and then analyses the results. The input to the simulator could be a sequence of input events or previously logged scenarios. From each input event and the current state, the simulator determines the next state [20].

For RSML, the check for consistency and completeness is elaborated on the AND/OR of the guarding conditions on the transitions [12, 13]. To check for consistency and completeness, every state must have a deterministic behavior, and the disjunction of the guarding consitions on the transitions should be a tautology.

The NIMBUS environment based on RSML^{-e} (Requirements State Machine Language without events) to validate system requirements was adopted. Simulation and execution are used to validate system's behavior. The Input/Output to the simulator could be stored and then analysed, or some test scenarios could be generated. The analyst could start with a user input or a simple model trying to imitate the environment actions. Then, as the evaluation progresses and the specification is more refined, the analyst could come up with more detailed models leading to better outcomes [47].

For the function tables [36], to check for consistency, the conditions of a function should not overlap. However, when it happens that there is an overlap, the output should be same for the overlapping conditions. For completeness, the disjunction of all the conditions of a function should be a tautology. (The validation process was not discussed).

To check the consistency and completeness of decision tables, Hoover and Chen developed a tool named "Tablewise" [22]. The consistency checking ensures that there is no overlap between different operational procedures. For that, the authors proposed a table containing the conjunction of the engagement criterion that belong to different operational procedures. Unfortunately, the resulted table may be big and also it is time consuming to check the overlap between all the operational procedures especially when the source table is large. For completeness, first the authors introduced a coverage table which consists of

the negation of the disjunction of the engagement criterion. However, such a table may be very big. So, the authors proposed what they called a "structured analysis" to localize flaws in the table preventing it from defining a function. Validation was not discussed.

2.8 Discussion

In this section, we depict the merits and disadvantages of the previously discussed techniques [5]. All of the discussed techniques are used to specify and analyse software systems. They rely on a tabular notation which is very helpful to increase readability. To achieve that, some of the techniques combine another notation with the tabular one. For instance, RSML is based on Harel statecharts, and it incorporates a graphical notation in addition to the tabular one. For function tables, besides the use of tables, sometimes macros are used to help shortening cells contents, and hence enabling the table not to be cumbersome.

It is very important to compose and decompose tables in a modular way. Specially for large systems, the big table used to represent the system's behaviour could be decomposed into smaller tables. For tabular expressions, the system's behaviour is represented by a relation that might be complex. That complex relation should be decomposed into smaller relations. In some cases, these relations are easily defined in some cells of the table. In other cases, a cell may refer to another table [28]. The concept of table composition and decomposition of tables is not adopted by the SCR method. Therefore, states and transitions are not decomposed. This issue might be a disadvantage of the method, especially while specifying large systems. For RSML, hierarchy is supported, and the AND/OR table decomposition is allowed. For function tables, the system is seen as a collection of tables without considering composition and decomposition of tables. In order to have a more compact notation, Hoover and Chen [22] proposed partitioned decision tables . They are

a kind of complex tables where the cells could be simple decision tables or part of simple decision tables.

In order to check for properties such as consistency and completeness, most of the techniques use a similar way to check for tautologies, except for decision tables (Hoover and Chen [22]), where the check for consistency and completeness is elaborated at a primitive level as previously discussed at section 2.7.

In the literature there are other table-based specification techniques that we did not include in our survey. The reason is that some of these specification techniques are mainly based on a graphical notation, and they do not have a well defined syntax and semantics, such as the StP (Structured Environment) technique [3]. There are other techniques that we did not conclude since they do not take into consideration the verification and validation for example the StP and VFSM (Virtual Finite State Machine) techniques [49].

Chapter 3

Tabular expressions vs Software Cost Reduction method

3.1 Introduction

Table-based specification techniques are readable and efficient. They allow us to express a system specification in a very compact and precise way. Moreover, they scale nicely when applied to practical software systems [19]. Among these table-based specification techniques, we highlight SCR and tabular expressions which have been and are still successfully used in practice.

The Software Cost Reduction (SCR) is a quite popular formal method to specify system requirements. It has been substantially improved and extended over the last decade [19]. However, its semantic is not very precise. Tabular expressions allow us to represent systems specifications in a compact and still precise manner using a multi-dimensional syntax [27]. Moreover, their intuitive semantics make them more suitable for most applications and easier for people who are not very familiar with the application domain. A comparison between the two methods can be found in [6].

$$f(x,y) = \begin{cases} 0 & \text{if} \quad x \geq 0 \wedge y = 10 \\ x & \text{if} \quad x < 0 \wedge y = 10 \\ y^2 & \text{if} \quad x \geq 0 \wedge y > 10 \\ -y^2 & \text{if} \quad x \geq 0 \wedge y < 10 \\ x + y & \text{if} \quad x < 0 \wedge y > 10 \\ x - y & \text{if} \quad x < 0 \wedge y < 10 \end{cases}$$

Figure 3.1: Mathematical notation of the function f

In this chapter we present the semantics of tabular expressions. Then, we present the algorithms we devised to convert SCR tables into tabular expressions. The transformations we use preserve the meaning of the original tables. Also, we give examples of such transformations. The converted SCR tables are quite readable and could be easily interpreted even by people who are not quiet familiar with the method. Later, we address the issue of improving SCR semantics. We present a new way to model the SCR events with first order logic. We also depict a simpler way to define SCR events with propositional logic. Finally, we give an illustrative example. There are many advantages that are gained by improving the SCR semantics. For instance, the task of verification and validation will be easier, and the SCR toolset supporting the method will be improved.

Section 3.2 is devoted to present tabular expressions semantics. In Section 3.3, we present the transformation of SCR tables into tabular expressions, and give simple examples to illustrate the conversion. Section 3.4 presents how we improve SCR semantics. In Section 3.5, we discuss the merits and disadvantages of tabular expressions and SCR.

3.2 Tabular expressions semantics

Tabular expressions are very convenient and readable especially for classic mathematical notation [25]. For that purpose, we illustrate an example borrowed from [25] to represent

$$
\begin{array}{rcl}
f(x,y) & = & \text{if} \quad x \geq 0 \wedge y = 10 \quad \text{then} \quad 0 \\
\text{else if} \quad x & < & 0 \wedge y > 10 \quad \text{then} \quad x \\
\text{else if} \quad x & \geq & 0 \wedge y > 10 \quad \text{then} \quad y^2 \\
\text{else if} \quad x & \geq & 0 \wedge y < 10 \quad \text{then} \quad -y^2 \\
\text{else if} \quad x & < & 0 \wedge y < 10 \quad \text{then} \quad x + y \\
\text{else if} \quad x & < & 0 \wedge y > 10 \quad \text{then} \quad x - y
\end{array}
$$

Figure 3.2: The function f defined with predicate logic

H_2

	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x + y$	$x - y$

H_1

Figure 3.3: The function f defined by a table

a function with different notations. For instance, the function $f(x, y)$ written in a mathematical notation is presented in Figure 3.1. Using predicate logic, the function $f(x, y)$ is presented in Figure 3.2. These two notations are not very readable although the function described is quite simple. It is evident that the function is much more readable with tabular notation as presented in Figure 3.3.

If tabular expressions have been successfully used for many years, it is because they have a precise and well-defined semantics, and this fact should not be underestimated. A lot of work has been done to develop the semantics of tabular expressions over the last two decades [51]. At the beginning, the semantics of tabular expressions was rather informal and emphasized on the uniform understanding of table functionality. For instance, Table 3.1 from older tabular expressions [51] was interpreted as:

	Result
Condition	f_name
Condition 1	res 1
Condition 2	res 2
... ...	...
Condition n	res n

Table 3.1: Simple function table from older tabular expressions

If Condition 1 *then* f_name = res 1

elseif Condition 2 *then* f_name = res 2

elseif ...

elseif Condition n *then* f_name = res n

It was also required that:

Condition i $\wedge$ Condition j $\Leftrightarrow$ False $\forall\, i, j = 1, ..., n \quad i \neq j$ (disjointness)

and

Condition 1 $\vee$ Condition 2 $\vee$ ... $\vee$ Condition n $\Leftrightarrow$ True (completeness)

Later on, additional requirements (e.g. disjointness and completeness) were removed. Nowadays, tables are checked for consistency and completeness with the disjointness and domain coverage properties. That is achieved while considering the structure of the table, and not its content.

Having formal semantics is fundamental for many reasons. For instance, the semantic is needed to either link a table and its sub-tables, or to link tables with different types. In both cases, if the table composition is done without a formal semantics, then the resulting table may not correspond to a consistent description of the system [28, 51]. Another reason is that tabular expressions have been proved to be invaluable for documenting software requirements and design [51, 53]. Such documents read by different users should converge to the same meaning of the table functionality, and in order to achieve that tabular expressions should have well defined semantics. Another strong motivation is that in order to be

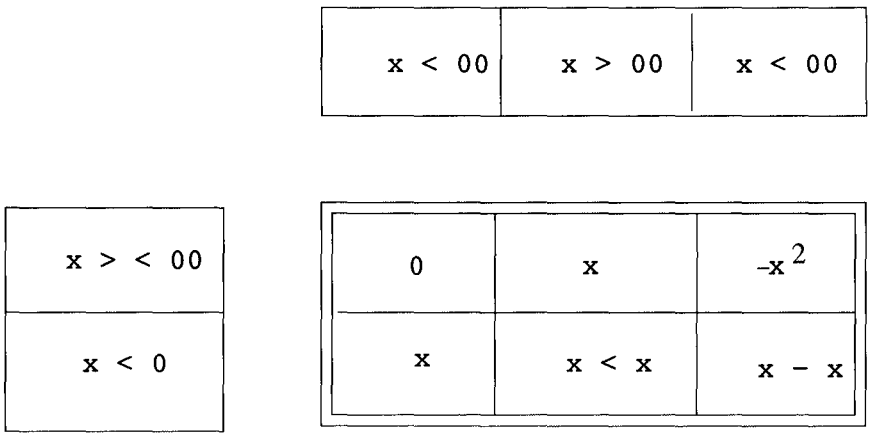


Figure 3.4: Stage 0. A normal table

used in practice, tabular expressions should be supported by tools for creating, editing, and transforming tables. Creating tools without a reasonable formal semantics may and often will lead to their failure [37].

To present the formal definition of tabular expressions and their semantics, we illustrate this in the following with an example of a normal function table at different stages of development borrowed from [51]. Stage 0 is shown in Figure 3.4. A normal function table is presented, where the elements of the headers are predicate expressions, and the elements of the grid G are terms. A multi-dimensional table is composed of *headers* $H_1, H_2, ..., H_n$, and one *grid* G. Both headers and the grid are built from ordered *cells* containing some expressions. The first stage is shown in Figure 3.5. There are two headers H_1 and H_2, and one grid G presented with a double border. The headers and the grid of a table T form what is called the *raw skeleton table* T^{raw}, which is the tuple $(H_1, H_2, ..., H_n, G)$. Then the flow of information amongst table components is determined to indicate *"where do I start reading the table and where do I get my result?"* (see Figure 3.6). Table cells are divided into two types: *guard cells* and *value cells*. Guard cells contain the predicates, while value cells contain the results. Value cells are often (but not always) represented

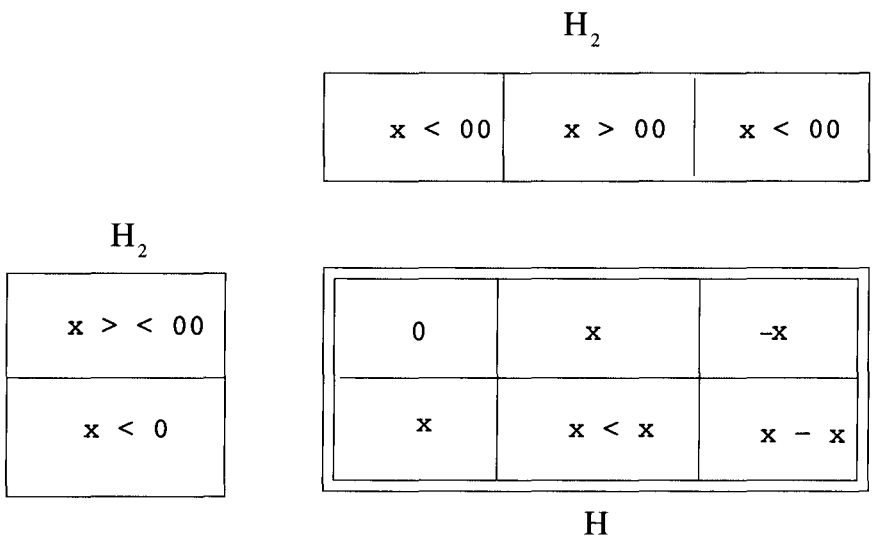


Figure 3.5: Stage 1. Assigning headers and a grid

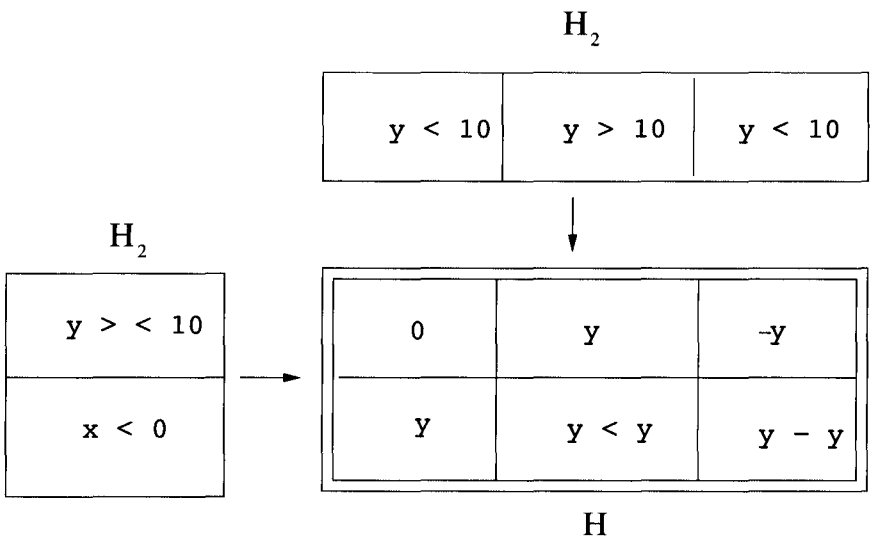


Figure 3.6: Stage 2. Adding information flow

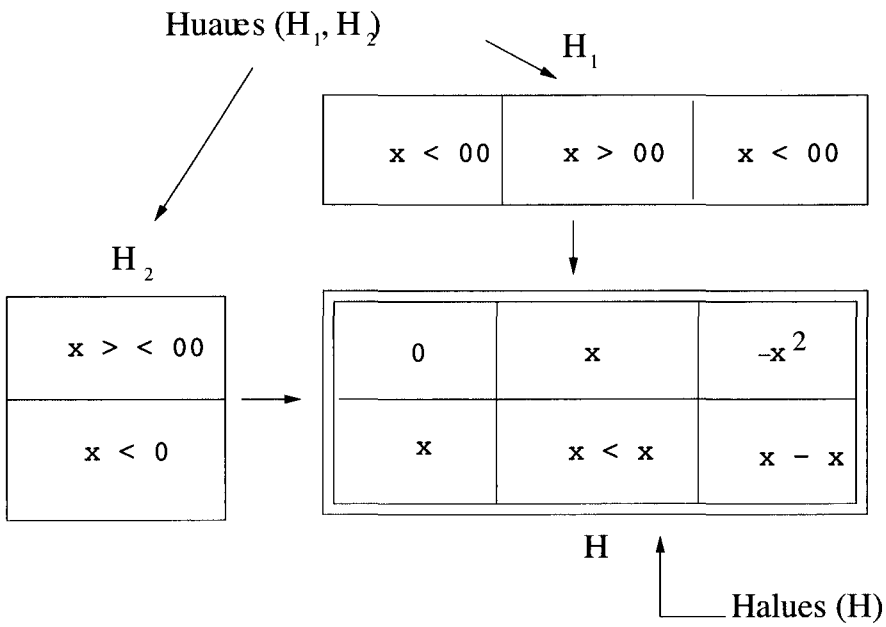


Figure 3.7: Stage 3. Identifying guards and value cells

with a double line border (see Figure 3.7). The information flow, the guard cells, and value cells represent what is called the *Cell Connection Graph* (CCG) of the tabular expression. CCG model information flow and provide a taxonomy of tabular expressions. Different CCG's correspond to different types of tables. For instance, the CCG from left hand side of Figure 3.8 corresponds to a normal table. The CCG is typically presented by an icon which resembles it and is usually placed at the top left corner of the table (see left part of Figure 3.10). After determining the CCG, the *medium table skeleton* T^{med} of the table is defined as the following tuple $T^{med} = (CCG, H_1, H_2, ..., H_n, G)$. Then, a *well done table skeleton* T^{well} of the table is defined as $T^{well} = (P_T, r_T, C_T, CCG, H_1, ..., H_n, G)$, where P_T is a table predicate rule indicating how predicates are to be built from the contents of table cells, r_T is a table relation rule indicating how relations/functions are to be built from the contents of table cells, C_T is a table composition rule which states how the global relation/function is built from local representations. The shape of C_T depends on the type

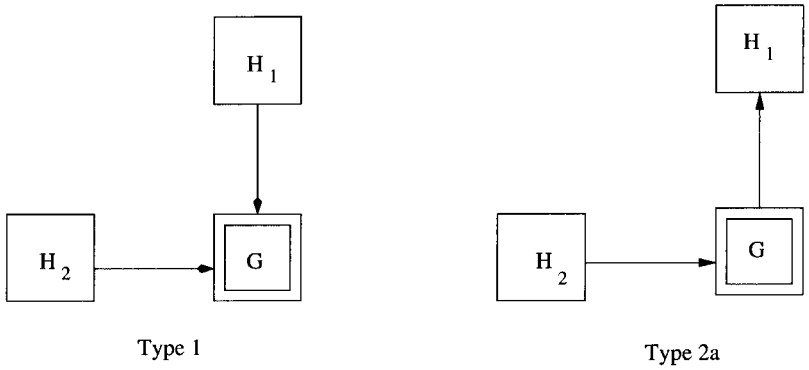
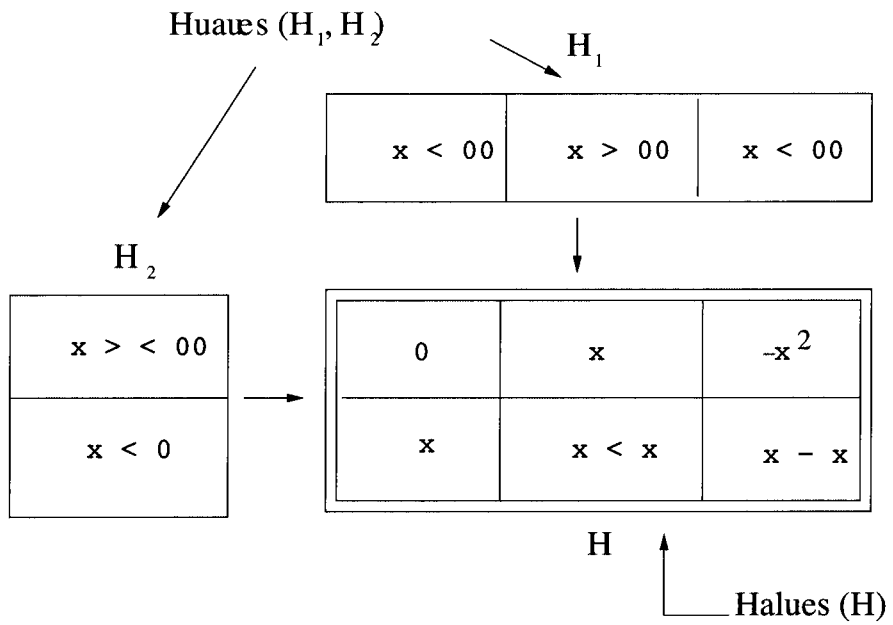


Figure 3.8: Stage 4. Examples of Cell Connection Graphs



$P_T = H_1 \wedge H_2$
$r_T = G$
$C_T : R = \bigcup_{i=1,2,3,j=1,2} R_{ij}$

Figure 3.9: Stage 5. Specifying predicate, relation and composition rule

of a table, i.e. CCG.

For the normal table presented in Figure 3.9, we have: $P_T = H_1 \wedge H_2$, $r_T = G$, while C_T is given by the relational formula $\cup_{i=1}^{3} \cup_{j=1}^{2} R_{ij}$, where R_{ij} is a relation corresponding to the expressions in the cells $H_1[i]$, $H_2[j]$ and $G[i,j]$. Finally, a tabular expression T is formally defined as:

$$T = (P_T, r_T, C_T, , CCG, H_1, ..., H_n, G, \Psi, IN, OUT)$$

where Ψ is a mapping assigning *predicate expressions* to guard cells, and *relation expressions* to value cells. IN is the set of inputs (e.g. $Reals \times Reals$), and OUT is the set of outputs. The predicate expressions have variables over IN, and the relation expressions have variables over $IN \times OUT$. The meaning (semantics) of a tabular expression T is given by a relation $R_T \subseteq IN \times OUT$, which is defined as:

$$R_T = C_T(R_\alpha).$$

For the normal table from Figure 3.10, $R_T = \cup_{i=1}^{2} \cup_{j=1}^{3} R_{ij}$, where R_{ij} are as defined above. In Section 4.2 more examples are provided.

Now to be able to distinguish the semantics difference between different kinds of tables, the CCG should be defined. The CCG could be depicted by an acyclic graph where the nodes are the headers and the grids, and the arcs are the information flow between the connected table components. The nodes are portioned into two classes which are *guarded components* (Guards (T)) and *value components* (Values (T)). Each arc must start from or end at the grid G [28]. Let Compt(T) = $\{H_1, H_2, \ldots, H_n, G\}$. A Cell Connection Graph is

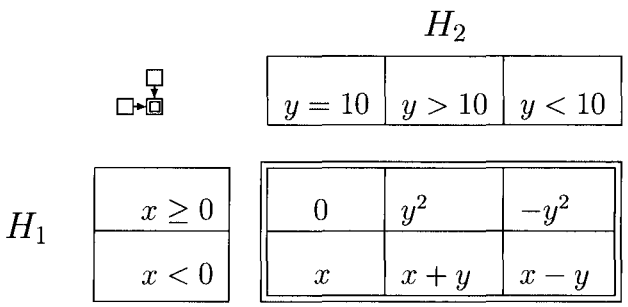


Figure 3.10: A normal table and its cell connection graph.

an asymmetric relation:

$$\longmapsto \subseteq \text{Compt (T)} \times \text{Compt (T)}$$

s.t. $\forall A, B \in$ Compt (T)

(3.1) $A \longmapsto B \Longrightarrow ((A = G \vee B = G) \wedge A \neq B)$

The relation $\overset{+}{\longmapsto}$ is the transitive closure[1] of $\longmapsto$.

The reflexive transitive closure[2] of the relation $\longmapsto$ is a partial order. A component $A \in$ Compt(T) is maximal if:

$$A \overset{*}{\longmapsto} B \Longrightarrow B = A \quad \forall B \in \text{Compt (T)}$$

[1] $A \overset{+}{\longmapsto} B \Longleftrightarrow (A \longmapsto B) \vee (\exists A_1, A_2, \ldots, A_k . A \longmapsto A_1 \longmapsto A_2 \longmapsto \ldots \longmapsto A_k \longmapsto B)$

[2] $A \overset{*}{\longmapsto} B \Longleftrightarrow (A = B) \vee (A \longmapsto B) \vee (\exists A_1, A_2, \ldots, A_k . A \longmapsto A_1 \longmapsto A_2 \longmapsto \ldots \longmapsto A_k \longmapsto B)$

A component $A \in$ Compt(T) is minimal if:

$$B \overset{*}{\longmapsto} A \Longrightarrow B = A \quad \forall B \in \text{Compt (T)}$$

A component $A \in$ Compt(T) is neutral if it is neither minimal nor maximal. The components containing cells describing domains cannot be maximal, and the components containing cells describing values of the relation/function cannot be minimal. Thus the partition of Comp(T) into Guards (T) and Values(T) should satisfy the following conditions:

$$\begin{aligned}
&\text{Compt(T)} = \text{Guards(T)} \cup \text{Values(T)} \\
&\text{Guards(T)} \cap \text{Values(T)} = \emptyset \\
&\text{A is maximal} \Longrightarrow A \in \text{Values(T)} \\
&\text{A is minimal} \Longrightarrow A \in \text{Guards(T)} \\
&\forall A \in \text{Guards(T)}. \forall B \in \text{Values(T)}. A \overset{+}{\longmapsto} B
\end{aligned} \tag{3.2}$$

Now the CCG is formally defined as a triple:

$$\text{CCG} = (\text{Guards(T)}, \text{Values(T)}, \longmapsto)$$

where, $\longmapsto$ satisfies condition 3.1, Guards(T) and Values(T) satisfies condition 3.2.

There are six different kinds of CCGs:

- Type 1: each element is either maximal or minimal, and there is only one maximal element. An example of a CCG of type 1 is shown in Figure 3.11.

- Type 2a: there is only one maximal element and one neutral element, and the neutral element belongs to Guards(T). An example is given in Figure 3.12.

- Type 2b: there is only one maximal element and one neutral element, and the neutral

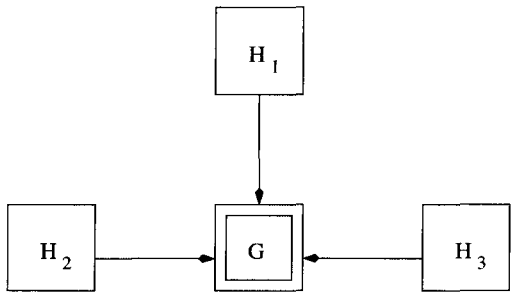


Figure 3.11: Type1. Each element is either maximal or minimal. There is only one maximal element.

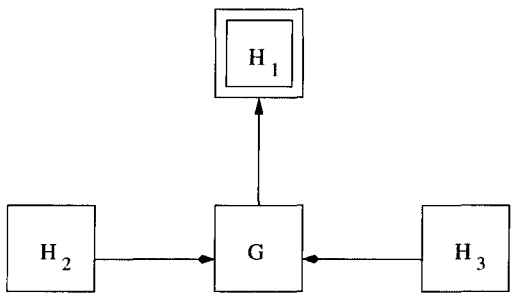


Figure 3.12: Type2a. There is only one maximal element and a neutral element. The neutral element belongs to Guards(T)

element belongs to Values(T). An example is presented in Figure 3.13.

- Type 3a: there is a neutral element and more than one maximal element, and the neutral element belongs to Guards(T). An example is depicted in Figure 3.14.

- Type 3b: there is a neutral element and more than one maximal element, and the neutral element belongs to Values(T). An example is illustrated in Figure 3.15.

- Type 4: each element is either maximal or minimal, and there is only one minimal element. An example is given in Figure 3.16.

The previous examples illustrating the six kinds of CCGs are borrowed from [28]. In the six examples the number of headers is equal to three. This classification introduced by

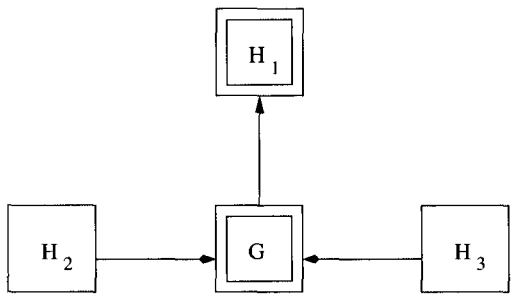


Figure 3.13: Type2b. There is only one maximal element and a neutral element. The neutral element belongs to Values(T)

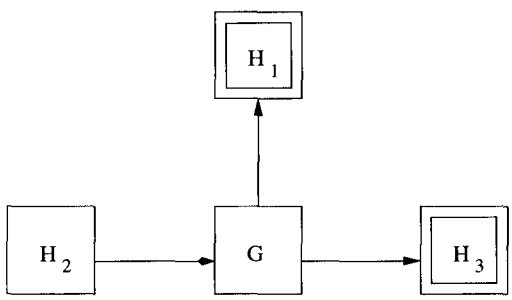


Figure 3.14: Type3a. There is a neutral element and more than one maximal element. The neutral element belongs to Guards(T)

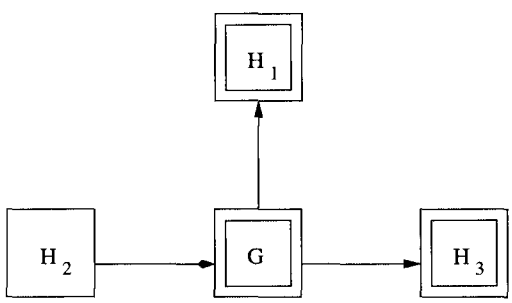


Figure 3.15: Type3b. There is a neutral element and more than one maximal element. The neutral element belongs to Values(T)

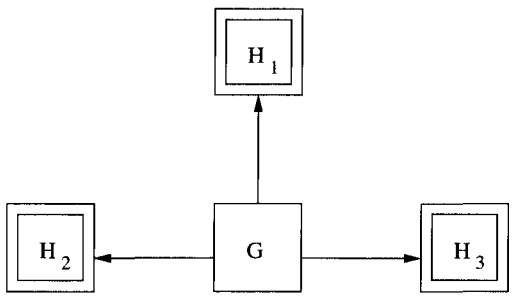


Figure 3.16: Type4. Each element is either maximal or minimal. There is only one minimal element

Janicki is application independent. It was proposed in [25], and then refined in [26, 28].

D.L. Parnas proposed the first semantics analysis of tabular expressions based on types of applications [38]. He suggested another classification for tabular expressions that is based on types of applications [38]. He identified ten classes of tables:

- Normal function tables: in a normal function table the elements of the headers are predicate expressions, and the elements of the grid G are terms.

- Inverted function tables: in an inverted function table T, the elements of the first header H_1 are terms. The elements of the other headers $H_2, \cdots, H_{dim(T)}$ are predicate expressions, and the elements of the grid G are predicate expressions as well.

- Vector function tables: in a vector function table the elements of the headers $H_1, H_3, \cdots, H_{dim(T)}$ are predicate expressions. The elements of the grid G are terms. The elements of the second header H_2 are variables.

- Normal relation tables: in normal relation table, the elements of the headers and the grid are predicate expressions. However there is a distinguished variables denoted by "®" that should not appear in the headers.

- Inverted relation tables: in an inverted relation table, the elements of the headers and

the grid are predicate expressions. The variables "®" may appear in the header H_1 but not in the headers $H_2, \cdots, H_{dim(T)}$ and the gird G.

- Vector relation tables: in a vector relation table the elements of the headers $H_1, H_3, \cdots, H_{dim(T)}$, and the elements of G are predicate expressions. The elements of H_2 are variables.

- Mixed vector tables: in a mixed vector table the elements of the headers $H_1, H_3, \cdots, H_{dim(T)}$ are predicate expressions. The elements of the second header H_2 are variables, and the elements of the grid G could be either predicate expressions or terms.

- Predicate expressions tables: in predicate expressions tables, the elements of all headers and the grid are predicate expressions. The table T could be seen as a predicate.

- Characteristic predicate tables: in a characteristic predicate table, the elements of the headers and the grid are predicate expressions. The table T could be seen as relation for which the domain and range contain tuples of a fixed length.

- Generalized decision tables: in a generalized decision table, the elements of the headers H_1 and H_2 are terms. The other headers $H_3, \cdots, H_{dim(T)}$ are not used in this class of tables. The elements of the grid G are predicate expressions that contain in some cases the symbol "#". In a cell the "#" symbol is replaced by its corresponding value.

Each class proposed by Parnas in [38] could be seen as a special case of the generic model proposed by Janicki [25, 26, 28]. For instance, Type 1 corresponds to normal tables. Type 2a corresponds to inverted, decision and generalized decision tables. Type 2b corresponds to vector tables. For types 3a, 3b and 4, we do not know any practical use for them so far [28].

Recently in [30], Jin and Parnas propose a new model for defining tabular expressions. Tabular expressions are seen as an indexed set of grids, and a grid is viewed as an indexed set of expressions. Tabular expressions are classified into table-types. A tabular expression is given a meaning only if it has a previously defined type. To define an expression type, a restriction schema, and evaluations schemas for tables are given. The restriction schema is a predicate presenting the restrictions imposed on the table. An evaluation schema defines equivalent expressions of a previously defined type. The model is applied on previous defined tables such as normal function tables, inverted tables, generalized decision tables, and new ones such as tables with redundant header grids, circular tables, and locator tables. The new types of tables introduced in this work might be of interest in practice.

3.3 Transforming SCR tables into tabular expressions

In this section, we illustrate how SCR tables can be transformed into function tables, and we provide examples for that. We present algorithms to convert SCR tables seen as two-dimensional arrays into function tables. Indeed, function tables are a special kind of tabular expressions. In Appendix B, we show how function tables fit in the general framework of tabular expressions.

3.3.1 Transforming Condition Tables

In the following we show how to convert SCR condition tables (CT) into vertical condition tables (VCT) of function tables. In SCR, a condition table (CT) defines a variable according to a mode and a condition. In vertical condition tables (VCT), the left bottom cell indicates the name of the function. For example, in Table 3.2, the name of the function is: f-SafetyInjection. The other bottom columns indicate the value of the function when the condition of the respective columns is True. This implies that each value of the function can

be obtained from a disjunction of the conjunctions of the respective cell and its respective mode value. This justifies our conversion given in Algorithm 3.1. Notice that in converting a CT table into a VCT table, the last row remains the same, and it contains the name of the function and its values. To have a simplified table, any expression of the form False $\vee$ *Expression* or True $\wedge$ *Expression* is replaced with *Expression*. The time complexity of this algorithm is $O(nm)$. As a result of the application of Algorithm 3.1 to convert the CT table drawn in Table 2.5, we obtain the VCT table presented by Table 3.2. Both of the two tables are behaviourally equivalent.

The condition table drawn in Table 3.3 defines the controlled variable *heat*. The controlled variable *heat* is $True$ if the variable *setting* is enabled and the variable *desired* is $True$. Otherwise, if *setting* is disabled, or if *setting* is enabled and *desired* is $False$, the variable *heat* is $False$. The example is borrowed from an SCR specification of a thermostat [44]. We also apply Algorithm 3.1 to convert the condition table defining the controlled variable *heat* presented in Table 3.3 into a vertical condition table. As a result of the conversion, we obtain table 3.4.

Algorithm 3.1 Converting CT tables into VCT tables

```
{n is the number of columns of the CT table}
for j from 1 to n − 1 do
  Sum ⟵ False
  {m is the number of rows of the CT table}
  for i from 1 to m − 1 do
    Sum ⟵ Sum ∨ (CT[i, 1] ∧ CT[i, j + 1])
  end for
  VCT[1,j] ⟵ Sum
end for
for j from 1 to n − 1 do
  VCT[2, j] ⟵ CT[m, j + 1]
end for
```

VCT: SafetyInjection

(Pressure = High) $\vee$ (Pressure = Permitted) $\vee$ (Pressure = TooLow) $\wedge$ (Overridden = $True$)	(Pressure = TooLow) $\wedge$ (Overridden = $False$)
f-SafetyInjection=Off	f-SafetyInjection=On

Table 3.2: Vertical condition table for *Safety Injection*

Setting = disabled	$True$	$False$
Setting = enabled	not-desired	desired
heat	$False$	$True$

Table 3.3: Condition table for *heat*

VCT: Heat

(Setting = disabled) $\vee$ (Setting = enabled) $\wedge$ (not-desired)	(Setting = enabled) $\wedge$ (desired)
(f-heat = $False$)	(f-heat = $True$)

Table 3.4: Vertical condition table for *heat*

VCT: Overridden

(Pressure=TooLow) $\vee$ (Pressure=Permitted) $\wedge$ (block =On $\wedge$ Reset = Off)	(Pressure=High) $\vee$ ((Pressure = TooLow) $\vee$ (Pressure=Permitted) $\vee$ (Reset = On))
f-Overridden= $True$	f-Overridden=$False$

Table 3.5: Vertical condition table for *Overridden*

3.3.2 Transforming Event Tables

An event table defines a variable according to a mode and an event. An event occurs when a condition value switches from $True$ to $False$. The notation "@T(c)" means that condition c becomes $True$, and the notation "@F(c)" denotes that condition c becomes $False$ [19]. For example, @T(Block=On) when T(Reset=Off) means that the operator turns Block from Off to On when the Reset is Off. In SCR, the notation @T(Inmode) means that the system enters into the class of modes in that row. In Table 2.6, the mode Pressure is defined via the current mode, and the events are defined on the variable WaterPress. Event tables and condition tables are behaviourally equivalent. In fact, in both tables a variable is defined via a mode and the change of a system state. Hence, event tables will also be transformed into vertical condition tables. Therefore, Algorithm 3.1 can be used to convert an SCR event table into a vertical condition table. As a result of applying it to convert the event Table 2.6 for *Overridden*, we obtain Table 3.5, which is behaviourally equivalent to it.

Another example is the event table presenting the latches of a measuring hydraulic system [16] which is presented by Table 3.6. The event table is interpreted as the following: The variable *tpressure-latch* is $True$ if both variables *mpressure-hold* and *tpressure-auto* are $True$, and it is $False$ if the variable *tpressure-auto* is $False$.

The result of the conversion of the event Table 3.6, is Table 3.7. Both tables are be-

Mode	Events	
	@T(mpressure-hold and tpressure-auto)	@F(tpressure-auto)
tpressure-latch	$True$	$False$

Table 3.6: Event table for *tpressure-latch*

VCT: *tpressure-latch*

(mpressure-hold = $True$) $\wedge$ (tpressure-auto=$True$)	(tpressure-auto=$False$)
tpressure-latch=$True$	tpressure-latch=$False$

Table 3.7: Vertical condition table for *tpressure-latch*

haviourally equivalent.

3.3.3 Transforming Mode Transition Tables

A mode transition table (MTT) generates a destination mode from a mode and an event. Similarly, a state transition table (STT) represents next state functions, given a current state and a condition. Modes are seen as classes of system states specifying the system behaviour. Therefore, the two tables are behaviourally equivalent. In the following we present an algorithm to show how mode transition tables (MTTs) are transformed into state transition tables (STTs). In this algorithm we start by filling the rows of the STT table with modes of the MTT table, and the columns of the STT table with events of the MTT table. Since each cell in STT can be indexed by a mode m and an event e as $T[m, e]$, we can easily compute the next state in STT as follows:

STT [MTT[i, 1],MTT[i, 2]] $\longleftarrow$ MTT[i, 3]

where MTT[$i, 1$] and MTT[$i, 2$] are the mode and event at row i, respectively. MTT[$i, 3$] designates the next mode at row i. Then, we have to merge the rows having identical states, and the columns having identical events to reduce the size of the table. The time complexity to look for rows with identical states, the columns having identical events, and merge them is $O(n^2)$. However, we were able to reduce the complexity to $O(n)$ with the indexing property of STTs tables that we proposed. In fact with the indexing property of STTs tables, we access directly to fill in the new states since they are indexed by mode and events. Finally, we fill the empty cells with the same current state to indicate that the state remains the same.

Algorithm 3.2 Converting MTT tables into STT tables

```
{initialize the first row of STT to the MTT input events starting at cell [1, 2]}
{n is the number of rows of MTT}
Let R be an empty set
for i from 1 to n do
   add MTT[i, 1] to R
end for
Copy the elements of R into the first row of STT starting a cell [1, 2]
{initialize the first column of STT to the MTT input modes starting at cell [2, 1]}
Let C be an empty set
for i from 1 to n do
   add MTT[i, 1] to C
end for
Copy the elements of C into the first column of STT starting a cell [2, 1]
for i from 1 to n do
   STT[1, i + 1] ← MTT[i, 2]
end for
for i from 2 to n do
   STT [MTT[i, 1], MTT[i, 2]] ← MTT[i, 3]
end for
{Handle STT's empty cells}
for every mode m and event e do
   if STT[m, e] is an empty cell then
      STT[m, e] ← m
   end if
end for
```

After applying algorithm 3.2, we find that Table 3.8 is behaviourally equivalent to Table 2.7. Also, Table 3.8 is more explicit than Table 2.7.

STT: f-Pressure

	@T(WaterPress $\geq$ Low)	@T(WaterPress $\geq$ Permit)	@T(WaterPress $<$ Low)	@T(WaterPress $<$ Permit)
TooLow	Permitted	TooLow	TooLow	TooLow
Permitted	Permitted	High	TooLow	Permitted
High	High	High	High	Permitted

Table 3.8: State transition table

We give another example borrowed from the SCR specification of the thermostat [44], where Table 3.9 is a mode transition table defining the mode class *setting*. It specifies whether the mode *setting* is enabled or disabled. We also apply algorithm 3.2, to convert the mode transition table 3.9 into the state transition table 3.10. As a result, the two tables are behaviourally equivalent.

3.4 Improving SCR semantics

In order to improve SCR semantics, and to avoid using ambiguous symbols (e.g. primed notations and prefixed notations with "@" symbols), we depict a more rigorous way to model SCR events with first order logic. In addition, we introduce a model for the first-order language. We also present a simpler way to define SCR events with propositional

Old Mode	**Event**	**New Mode**
disabled	@T(switch-on)	enabled
enabled	@T(not-switch-on)	disabled

Table 3.9: Mode transition table for *setting*

STT: *setting*

	@T(switch-on)	@T(not-switch-on)
disabled	enabled	disabled
enabled	disabled	enabled

Table 3.10: State transition table for *setting*

logic. Then, we give an illustrative example.

In SCR, a *Simple Condition* is either True, False, or a logical statement of the form $r \odot v$, where r belongs to the set of entity names, $\odot$ is a relational operator [3], and v belongs to the type of r. The set of entity names contains mode class names, input and output variables names, and term names. A *Condition* is a logical statement composed of *Simple Conditions* connected by logical connectors.

A *Basic Event* is denoted by @T(c), where c is a simple condition meaning that condition c becomes $True$. Similarly, @F(c) means that condition c becomes $False$.

A *Simple Conditioned Event* is denoted by @T(c) WHEN d, where d is a condition. It is defined as the following:

@T(c) WHEN d = c' $\wedge \neg c \wedge d$,

where the unprimed condition represents its old value, and the primed condition depicts its new value. Finally, a *Conditioned Event* is composed of Simple Conditioned Events connected by logical connectors.

Let us consider an example borrowed from [14]. The set of entity names denoted RF is defined by:

RF = {Block, Reset, WaterPres, Pressure, SafetyInjection, Overridden}.

[3] $\odot \in \{=, \neq <, >, \leq, \geq\}$

The type definitions are :

TY(Pressure) = {TooLow, Permitted, High}

TY(WaterPres) = {0, 1, 2, ..., 2000}

TY(Overridden) = $\{True, False\}$.

The conditions are specified on the values of entities in RF. For example, the conditioned event @T(Block=On) WHEN Reset=Off can be rewritten as :

$$\text{Block' = On} \wedge \text{Block = Off} \wedge \text{Reset = Off}$$

3.4.1 Event Modelling in First-Order Logic

Let $L = (\mathcal{C}, \mathcal{F}, \mathcal{P})$ be a first-order language such that $\mathcal{C}$ is a set of symbols called constants, $\mathcal{F}$ is a set of symbols called function symbols, each with arity ≥ 1, and $\mathcal{P}$ is a set of symbols called predicate symbols, each with arity ≥ 1. $\mathcal{P}$ contains the binary predicate symbol "=". Let $\mathcal{V}$ be the set of symbols called variables. $\mathcal{V}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{P}$ are pairwise disjoint.

A *term* in L is defined recursively as follows: a constant or a variable is a term. Given a function symbol f with n-arity and the terms $t_1, \ldots, t_n$, $f(t_1, \ldots, t_n)$ is a term.

A *formula* is defined as follows. Given a predicate P with n-arity, $t_1, \ldots, t_n$ terms, $P_n(t_1, \ldots, t_n)$ is a formula. $\neg F$, and $\forall x F$, $\exists x F$ are formula. Given two formulae F_1 and F_2, $F_1 \wedge F_2$ and $F_1 \vee F_2$ are formulae.

Given a language $L = (\mathcal{C}, \mathcal{F}, \mathcal{P})$ defined as above, we extend it with two symbols $pred$ and $succ$. The symbol $pred$ is needed to represent the past value of a term, and the symbol $succ$ is needed to know the future value of a given term.

The language L equipped with the two symbols $pred$ and $succ$ is said to be *event-enabled*. In fact, with our event-enabled language we are able to model the SCR system in which the environment generates a sequence of events, and the system reacts to these events by changing their states.

The terms $pred(t)$ and $succ(t)$ are defined as follows:

(3.1) **Definition.**

- If t is a constant, we have:

$$pred(t) = t$$

and,

$$succ(t) = t.$$

- If t is a variable, then $pred(t)$ and $succ(t)$ are both atomic terms.

(3.2) **Definition.**

- For a term t of the form $f_n(t_1, \ldots, t_n)$, we have:

$$pred((f_n)(t_1, \ldots, t_n)) = f_n(pred(t_1), \ldots, pred(t_n))$$

and,

$$succ((f_n)(t_1, \ldots, t_n)) = f_n(succ(t_1), \ldots, succ(t_n)).$$

(3.3) **Definition.**

- For a formula F of the form $(t_1 = t_2)$, we have:

$$pred(t_1) = pred(t_2)$$

and,

$$succ(t_1) = succ(t_2).$$

(3.4) **Definition.**

– For a formula F of the form $(P_m(t_1, \ldots, t_m))$, we have:

$$pred((P_m)(t_1, \ldots, t_m)) = P_m(pred(t_1), \ldots, pred(t_m))$$

and,

$$succ((P_m)(t_1, \ldots, t_m)) = P_m(succ(t_1), \ldots, succ(t_m)).$$

(3.5) **Definition.**

For the formulas F_1 and F_2 we have:

$$\neg(pred(F_1)) = pred(\neg(F_1))$$

$$pred(F_1 \vee F_2) = pred(F_1) \vee pred(F_2)$$

$$pred(F_1 \wedge F_2) = pred(F_1) \wedge pred(F_2)$$

and,

$$\neg(succ(F_1)) = succ(\neg(F_1))$$

$$succ(F_1 \vee F_2) = succ(F_1) \vee succ(F_2)$$

$$succ(F_1 \wedge F_2) = succ(F_1) \wedge succ(F_2).$$

(3.6) **Definition.**

– For every natural number n, we define the $pred^n$ symbol as follows:

$$\text{If} \quad n = 0, \quad \text{then} \quad pred^n(t) = t$$

$$\text{If} \quad n \geq 1, \quad \text{then} \quad pred^n(t) = pred(pred^{n-1}(t)).$$

– Similary, for every natural number n, we define the $succ^n$ symbol as follows:

$$\text{If} \quad n = 0, \quad \text{then} \quad succ^n(t) = t.$$

$$\text{If} \quad n \geq 1, \quad \text{then} \quad succ^n(t) = succ(succ^{n-1}(t)).$$

(3.7) **Definition.**

An event on a condition c is given by:

$$succ(c) \wedge \neg c$$

or equivalently:

$$c \wedge \neg pred(c).$$

(3.8) **Definition.**

– For any term t:

$$pred(succ(t)) = t$$

and,

$$succ(pred(t)) = t.$$

With our definitions, we avoided the prefixed SCR notations with "@" to represent events by @T(c) or @F(c). Hence, it becomes straightforward to express @F(c) in terms of $pred$ and $succ$. In fact, it is given by the forumla:

$$\neg c \wedge pred(c)$$

or equivalently:

$$\neg succ(c) \wedge c.$$

(3.9) **Definition.**

An *n-past event on a variable* x denoted by $event_n^-(x)$ is defined by:

$$\neg(pred^n(x) = x).$$

An *n-future event on a variable* x denoted by $event_n^+(x)$ is defined by:

$$\neg(succ^n(x) = x).$$

An n-past (respectively n-future) event indicates if an event happens between the current and the n-past (respectively n-future) time.

(3.10) **Definition.** Let f be a function symbol with k-arity, and k terms $t_1, \ldots, t_k$. We call an *n-past event* of the function f, the formula denoted by $event_n^-(f)(t_1, \ldots, t_k)$, and defined by:

$$\neg(pred^n(f(t_1, \ldots, t_k)) = f(t_1, \ldots, t_k)).$$

An *n-future event on the function* f is the formula denoted by $event_n^+(f)(t_1, \ldots, t_k)$, and defined by:

$$\neg(succ^n(f(t_1, \ldots, t_k)) = f(t_1, \ldots, t_k)).$$

As an example of 1-past event, let "inc" be increment function which adds the value one to any input value. $event_1^-(\text{inc})(x = 1)$ is defined by:

$$\begin{aligned} & \neg(pred(inc(x = 1)) = inc(x = 1)) \\ \iff \quad & \neg(pred(x = 2) = (x = 2)) \\ \iff \quad & True \end{aligned}$$

This indicates that there is a 1-past event since the value of the variable x changed.

(3.11) **Definition.** Given a predicate symbol Q with q-arity, and q terms $t_1, \ldots, t_q$, an *n-past event on predicate Q* is the formula $event_n^-(Q)(t_1, \ldots, t_q)$ defined by:

$$\neg(pred^n(Q(t_1, \ldots, t_k))) \wedge Q(t_1, \ldots, t_k).$$

An *n-future event on predicate Q* is the formula $event_n^+(Q)(t_1, \ldots, t_k)$ given by:

$$\neg(succ^n(Q(t_1, \ldots, t_k))) \wedge Q(t_1, \ldots, t_k).$$

From the above definitions, we have the following lemma.

Lemma 3.4.1 *Given a function symbol f with k-arity and k terms $t_1, \ldots, t_k$, we have:*

$$f(pred^n(t_1), \ldots, pred^n(t_k)) = pred^n(f(t_1, \ldots, t_k)).$$

PROOF. By induction:

1) Let us verify that it is True for the base case $n = 0$

$$\begin{aligned} & f(pred^0(t_1), \ldots, pred^0(t_k)) = pred^0((f(t_1, \ldots, t_k)) \\ \Longleftrightarrow \quad & < \text{Definition 6} > \\ & f(t_1, \ldots, t_k) = f(t_1, \ldots, t_k) \\ \Longleftrightarrow \quad & True \end{aligned}$$

2) Let us assume that for some n, we have:

$$f(pred^n(t_1), \ldots, pred^n(t_k)) = pred^n(f(t_1, \ldots, t_k)).$$

3) Let us prove that:

$$f(pred^{n+1}(t_1), \ldots, pred^{n+1}(t_k)) = pred^{n+1}(f(t_1, \ldots, t_k)).$$

$$\begin{aligned}
& f(pred^{n+1}(t_1), \ldots, pred^{n+1}(t_k)) \\
= \quad & < \text{By Definition 6} > \\
& f(pred(pred^n(t_1), \ldots, pred(pred^n(t_k)) \\
= \quad & < \text{By Definition 2} > \\
& pred(f(pred^n(t_1), \ldots, pred^n(t_k)) \\
= \quad & < \text{By induction hypothesis} > \\
& pred(pred^n(f(t_1, \ldots, t_k))) \\
= \quad & < \text{By Definition 6} > \\
& pred^{n+1}(f(t_1, \ldots, t_k))
\end{aligned}$$

□

Lemma 3.4.2 *Given a function symbol f with k-arity and k terms $t_1, \ldots, t_k$, we have:*

$$f(succ^n(t_1), \ldots, succ^n(t_k)) = succ^n(f(t_1, \ldots, t_k)).$$

PROOF. By induction:

1) Let us verify that it is True for the base case $n = 0$

$$
\begin{aligned}
& f(succ^0(t_1), \ldots, succ^0(t_k)) = succ^0((f(t_1, \ldots, t_k)) \\
\Longleftrightarrow \quad & < \text{Definition 6} > \\
& f(t_1, \ldots, t_k) = f(t_1, \ldots, t_k) \\
\Longleftrightarrow \quad & True
\end{aligned}
$$

2) Let us assume that for some n, we have:

$$f(succ^n(t_1), \ldots, succ^n(t_k)) = succ^n(f(t_1, \ldots, t_k)).$$

3) Let us prove that:

$$f(succ^{n+1}(t_1), \ldots, succ^{n+1}(t_k)) = succ^{n+1}(f(t_1, \ldots, t_k)).$$

$$
\begin{aligned}
& f(succ^{n+1}(t_1), \ldots, succ^{n+1}(t_k)) \\
= \quad & < \text{By Definition 6} > \\
& f(succ(succ^n(t_1), \ldots, succ(succ^n(t_k)) \\
= \quad & < \text{By Definition 2} > \\
& succ(f(succ^n(t_1), \ldots, succ^n(t_k)) \\
= \quad & < \text{By induction hypothesis} > \\
& succ(succ^n(f(t_1, \ldots, t_k))) \\
= \quad & < \text{By Definition 6} > \\
& succ^{n+1}(f(t_1, \ldots, t_k))
\end{aligned}
$$

□

(3.12) **Definition.**

Given a formula F, an *F-guarded n-past event* on variable x denoted by $x|F$ is defined by:

$$event_n^-(x) \wedge F$$

An *F-guarded n-past event* on function symbol f denoted by $f|F$ is defined by:

$$event_n^-(f) \wedge F$$

An *F-guarded n-past event* on predicate symbol P denoted by $P|F$ is defined by:

$$event_n^-(P) \wedge F$$

(3.13) **Definition.**

Given a formula F, an *F-guarded n-future event* on variable x denoted by $x||F$ is defined by:

$$event_n^+(x) \wedge F$$

An *F-guarded n-future event* on function symbol f denoted by $f||F$ is defined by:

$$event_n^+(f) \wedge F$$

An *F-guarded n-future event* on predicate symbol P denoted by $P||F$ is defined by:

$$event_n^+(P) \wedge F$$

3.4.2 Model for the FOL L

A model M for the first-order language L is a two tuple (D, I), where D is a nonempty domain of individuals, and I is an interpretation function that assigns each constant, function symbol, and predicate symbol in L over D.

Let Var be the set of variables, $Term$ be the set of terms of L, $Form$ be the set of formulas of L, and Int be the set of integers.

A variable assignment into M is a total function such that:

$$\varphi : Var \times Int \rightarrow D$$

That is, a variable assignment into M maps each variable at some point in time to a value in D.

Let $VarAssign(M)$ be the set of all variable assignment into M.

The valuation function for M is the function:

$$V : (Term \cup Form) \times VarAssign \times Int \rightarrow D \cup \{true, false\}$$

It is defined by the following statements where $\varphi \in VarAssign$ and $i \in Int$:

- If x is a variable,

$$V(x, \varphi, i) = \varphi(x, i)$$

- If c is an individual constant,

$$V(c, \varphi, i) = I(c)$$

– If f is an n-ary function symbol and $t_1, ...t_n$ are terms of L,

$$V(f(t_1, ...t_n), \varphi, i) = I(f)(V(t_1, \varphi, i), ..., V(t_n, \varphi, i))$$

– If P is an n-ary predicate and $t_1, \cdots, t_n$ are terms of L,

$$V(P(t_1, \cdots, t_n), \varphi, i) = I(P)(V(t_1, \varphi, i), \cdots, V(t_n, \varphi, i))$$

– If t is a term of L,

$$V(pred(t), \varphi, i) = V(t, \varphi, i-1)$$

$$V(succ(t), \varphi, i) = V(t, \varphi, i+1)$$

– If t_1 and t_2 are terms of L,

$$V(t_1 = t_2, \varphi, i) = true \quad \text{if } V(t_1, \varphi, i) = V(t_2, \varphi, i)$$

$$V(t_1 = t_2, \varphi, i) = false \quad \text{otherwise.}$$

– If F is a formula of the form $P(t_1, \cdots, t_n)$, where P is an n-ary predicate and $t_1, ...t_n$ are terms of L,

$$V(\neg F, \varphi, i) = true \text{ if } V(F, \varphi, i) = false$$

$$V(\neg F, \varphi, i) = false \text{ otherwise}$$

– If F_1 is a formula of the form $P(t_1, \cdots, t_n)$, where P is an n-ary predicate and $t_1, ...t_n$ are terms of L, F_2 is a formula of the form $P(t_1, \cdots, t_m)$, where P is an

m-ary predicate and $t_1, \cdots, t_m$ are terms of L,

$$V(F_1 \vee F_2, \varphi, i) = false \text{ if } V(F_1, \varphi, i) = false \wedge V(F_2, \varphi, i) = false$$

$$\text{otherwise } V(F_1 \vee F_2, \varphi, i) = true$$

- If F_1 is a formula of the form $P(t_1, \cdots, t_n)$, where P is an n-ary predicate and $t_1, \cdots, t_n$ are terms of L, F_2 is a formula of the form $P(t_1, \cdots, t_m)$, where P is an m-ary predicate and $t_1, \cdots, t_m$ are terms of L,

$$V(F_1 \wedge F_2, \varphi, i) = true \text{ if } V(F_1, \varphi, i) = true \wedge V(F_2, \varphi, i) = true$$

$$\text{otherwise } V(F_1 \wedge F_2, \varphi, i) = false$$

- If F is a formula of the form $\forall x(P(t_1, \cdots, t_n))$, where P is an n-ary predicate and $t_1, \cdots, t_n$ are terms of L,

$$V(F, \varphi, i) = true \text{ if } V(F, \varphi[x \rightarrow y], i) = true \ \forall y \in D$$

$$\text{otherwise } V(F, \varphi, i) = false$$

- A formula A is valid in M if, for $\forall \varphi \in VarAssign$ and $i \in Int$,

$$V(A, \varphi, i) = true.$$

Lemma 3.4.3

$$\forall \textit{ terms } t \textit{ of } L, V(pred(succ(t)), \varphi, i) = V(t, \varphi, i).$$

PROOF.

$$V(pred(succ(t)), \varphi, i)$$

$$= V(succ(t), \varphi, i - 1)$$

$$= V(t, \varphi, i - 1 + 1)$$

$$= V(t, \varphi, i).$$

□

Lemma 3.4.4

$$\forall \textit{ terms } t \textit{ of } L, V(succ(pred(t)), \varphi, i) = V(t, \varphi, i).$$

PROOF.

$$V(succ(pred(t)), \varphi, i)$$

$$= V(pred(t), \varphi, i + 1)$$

$$= V(t, \varphi, i + 1 - 1)$$

$$= V(t, \varphi, i).$$

□

In some cases, we do not need all the power of first-order logic, and we simply could adopt propositional logic. Hence, in the following subsection we present a simpler way to model SCR events in propositional logic, where the conditions are simply propositions.

3.4.3 Event Modeling in Propositional Logic

Let $L = (\mathcal{P}, \neg, \wedge, \vee)$ be a propositional logic, such that $\mathcal{P}$ is a set of primitive symbols called atomic formulae, $\neg$ is a unary operator, $\wedge$ and $\vee$ are two binary operators. The set of propositions $\{\neg, \wedge, \vee\}$ is complete since every truth function can be represented by a formula using only members of this set.

A formula is defined as follows:

- A symbol P in $\mathcal{P}$ is a formula.
- Given two formulae F_1 and F_2, $\neg F_1$, $F_1 \wedge F_2$ and $F_1 \vee F_2$ are formulae.

Given a propositional language $L = (\mathcal{P}, \neg, \wedge, \vee)$, we extend it using two unary operators $pred$ and $succ$ such that for every two formulae P and Q we have:

- $pred(\neg P) = \neg pred(P)$.
- $pred(P \wedge Q) = pred(P) \wedge pred(Q)$.
- $pred(P \vee Q) = pred(P) \vee pred(Q)$.
- $succ(\neg P) = \neg succ(P)$.
- $succ(P \wedge Q) = succ(P) \wedge succ(Q)$.
- $succ(P \vee Q) = succ(P) \vee succ(Q)$.
- $pred(succ(P)) = P$.
- $succ(pred(P)) = P$.

We define $succ$ similarly.

(3.14) **Definition.** For a natural number n, the unary operator $pred^n$ is defined recursively as follows:

- $pred^0(P) = P$.
- $pred^n(P) = pred(pred^{n-1}(P))$.
- $succ^0(P) = P$.
- $succ^n(P) = succ(succ^{n-1}(P))$.

(3.15) **Definition.** An n-past event on formula F denoted by $event_n^-(F)$ is defined:

$$\neg(pred^n(F) \wedge F).$$

An n-future event on formula F denoted by $event_n^+F$ is defined by:

$$\neg(succ^n(F) \wedge F).$$

(3.16) **Definition.** A formula R, an R-guarded n-past event on formula F denoted by $P|R$ is defined by:

$$event_n^-(F) \wedge R.$$

An R-guarded n-future event on formula F denoted by $P||R$ is defined by:

$$event_n^+(F) \wedge R.$$

If we denote $\neg P \vee Q$ by $P \Rightarrow Q$, then,

$$pred(P \Rightarrow Q) \Leftrightarrow (pred(P) \Rightarrow pred(Q)).$$

Similarly,

($\neg pred$(Pressure=High)) $\wedge$ (Pressure=High) $\vee$ ($\neg pred$(Pressure = Permitted)) $\wedge$ (Pressure = Permitted) $\vee$ ($\neg pred$(Pressure=TooLow)) $\wedge$ (Pressure=TooLow) $\wedge$ ($\neg pred$(Overridden = $True$)) $\wedge$ (Overridden = $True$)	($\neg pred$(Pressure=TooLow)) $\wedge$ (Pressure=TooLow) $\vee$ ($\neg pred$(Overridden = $False$)) $\wedge$ (Overridden = $False$)
($\neg pred$(SafetyInjection=Off)) $\wedge$ (SafetyInjection=Off)	($\neg pred$(SafetyInjection=On)) $\wedge$ (SafetyInjection=On)

Table 3.11: Vertical Condition Table for *SafetyInjection* with the "pred" symbol.

($succ$(Pressure=High)) $\wedge\neg$(Pressure=High) $\vee$ ($succ$(Pressure = Permitted)) $\wedge\neg$ (Pressure = Permitted) $\vee$ ($succ$(Pressure=TooLow)) $\wedge\neg$ (Pressure=TooLow) $\wedge$ ($succ$(Overridden = $True$)) $\wedge\neg$ (Overridden = $True$)	($succ$(Pressure=TooLow)) $\wedge\neg$ (Pressure=TooLow) $\vee$ ($succ$(Overridden = $False$)) $\wedge\neg$ (Overridden = $False$)
($succ$(SafetyInjection=Off)) $\wedge\neg$ (SafetyInjection=Off)	($succ$(SafetyInjection=On)) $\wedge\neg$ (SafetyInjection=On)

Table 3.12: Vertical condition table for *SafetyInjection* with the "succ" symbol.

$$succ(P \Rightarrow Q) \Leftrightarrow (succ(P) \Rightarrow succ(Q)).$$

Also for $P \Leftrightarrow Q$, we have:

$$pred(P \Leftrightarrow Q) \Leftrightarrow (pred(P) \Leftrightarrow pred(Q)).$$

Similarly,

$$succ(P \Leftrightarrow Q) \Leftrightarrow (succ(P) \Leftrightarrow succ(Q)).$$

$False$	($\neg pred$(Pressure = High)) $\wedge$ (Pressure = High)
($\neg pred$(Pressure=TooLow)) $\wedge$ (Pressure=TooLow) $\vee$ ($\neg pred$(Pressure = Permitted)) $\wedge$ (Pressure = Permitted) $\wedge$ ($\neg pred$(Block=On)) $\wedge$ (Block=On) $\wedge$ ($\neg pred$(Reset=Off)) $\wedge$ (Reset=Off)	($\neg pred$(Pressure=TooLow)) $\wedge$ (Pressure=TooLow) $\vee$ ($\neg pred$(Pressure = Permitted)) $\wedge$ (Pressure = Permitted) $\vee$ ($\neg pred$(Reset=On)) $\wedge$ (Reset=On)
($\neg pred$(Overridden)) $\wedge$ (Overridden)	($\neg pred$($\neg$ Overridden)) $\wedge$ ($\neg$ Overridden)

Table 3.13: Vertical condition table for *Overridden* with the "pred" symbol.

False	(*succ*(Pressure = High)) ∧ ¬ (Pressure = High)
(*succ*(Pressure=TooLow)) ∧¬ (Pressure=TooLow) ∨ (*succ*(Pressure = Permitted)) ∧ ¬ (Pressure = Permitted) ∧ (*succ*(Block=On)) ∧ ¬ (Block=On) ∧ (*succ*(Reset=Off)) ∧ ¬ (Reset=Off)	(*succ*(Pressure=TooLow)) ∧ ¬ (Pressure=TooLow) ∨ (*succ*(Pressure = Permitted)) ∧ ¬ (Pressure = Permitted) ∨ (*succ*(Reset=On)) ∧ ¬ (Reset=On)
(*succ*(Overridden)) ∧ (¬ Overridden)	(*succ*(Overridden)) ∧ (¬ Overridden)

Table 3.14: Vertical condition table for *Overridden* with the "succ" symbol.

3.4.4 Illustrative example

In this section, we provide examples where we transform SCR tables into new tables using the *pred* and *succ* symbols. We convert the condition Table 2.5 into Table 3.11 using the *pred* symbol. We also convert Table 2.5 into Table 3.12 using the *succ* symbol. All of these tables are behaviourally equivalent.

Also we transformed Table 2.6 into Table 3.13, and Table 3.14 with the *pre* and *succ* symbols respectilvely. Table 3.13 and Table 3.14 are behaviourally equivalent with the difference that we used the *pred* symbol in Table 3.13, and the *succ* symbol in Table 3.14.

Besides, we converted the Mode Transition Table 2.7 into State Transition Tables with the *pred* and *succ* symbols presented respectively in Tables 3.15 and 3.16 as well.

In SCR tables, there are some ambiguous notations that are not easily understood by users/developers who are not quiet familiar with the SCR method. Also, for SCR condition tables and event tables, it is not obvious that a variable is obtained by the conjunction of the respective mode and event. In our transformed tables, we removed the Mode column, and we write explicitly the mode with its respective event. In addition, with the two extra symbols *succ* and *pred*, we improve SCR events definition. In fact, for any event, we are able to determine all its predecessors and successors events. Besides, we avoid the fuzziness of the notation used in SCR to denote that the unprimed variables represent the old values of the variables, and the primed variables represent the new ones. From the examples that we provided, it is clear that our transformed tables are more readable and easier to interpret than SCR tables.

3.5 Discussion

In this section we present the merits and disadvantages of the two methods. For SCR, it relies on a state-based model. Some constructs were added to the general SCR model trying to make the specification more concise. Hence, the developers should be knowledgeable about those concepts, and this makes the task of building the specification difficult. We support our claim by a study on the application of SCR on a Space Station Biological Research Project at the NASA Ames Research Center (ARC) [43]. As a matter of fact, building the SCR specification took a lot of time. Also, the SCR project team had to intervene, and use their expertise about mathematical models and state machine models to help building the initial SCR specification, and then give it to the ARC developers for modification and extension.

Tabular expressions are based on a relational model, and relations are easily adopted by a wide range of people, even those who do not have a solid mathematical background. Their structure helps to make many manipulations such as checking whether a group of tables are behaviourally equivalent to another group of tables [27], or transforming a table into another simplest table [51]. With tabular expressions, it is easy to handle some mathematical manipulations, and also to check the consistency and completeness [28, 51]. Also with tabular expressions it is easy to build specification. For tabular expressions instead of having different kinds of tables, R. Janicki proposed a general framework unifying all kinds of tables previously proposed by D.L. Parnas, and where each kind of table could be seen as a special case of the general framework. For the SCR method, it is based on three kinds of simple tables and some constructs that were added to the language to specify system requirements.

It is very important to compose and decompose tables in a modular way. Unfortunately, the concept of table composition and decomposition of tables is not adopted by the SCR

method. Therefore, states and transitions are not decomposed. This issue might be a disadvantage of the method, especially while specifying large systems. For tabular expressions, the system's behaviour is represented by a relation that might be complex. That complex relation should be decomposed into smaller relations. In some cases, these relations are defined easily in some cells of the table. In other cases, a cell may refer to another table. Hence, we can compose and decompose tables in a modular way.

For the SCR method, it is quiet popular and it has been used and experimented in many industrial and academic organizations. However, it has some limitations regarding its semantics which is quite intuitive [19]. Also, the symbols adopted by the method are ambiguous. Moreover, the composition and decomposition of tables is not supported by the SCR method, and causing scalability problems of the method for large systems. In this work, we converted SCR tables into function tables, and in Appendix B, we showed how function tables fit in the general framework of tabular expressions. The transformation proposed is quite efficient and easy to implement. Many advantages are obtained with the conversion that we proposed. The tables are more readable, and could be easily interpreted even by people who are very knowledgeable of the domain. Besides, we avoid previous ambiguous symbols (e.g. primed notations and prefixed notations with "@" symbols). Hence, by improving SCR semantics, there are many tasks that could be carried out such as facilitating the verification and validation process, and improving the toolset supporting the SCR method.

STT: f-Pressure

	$\neg pred$ (WaterPress $\geq$ Low) $\wedge$ (WaterPress $\geq$ Low)	$\neg pred$ (WaterPress $\geq$ Low) $\wedge$ (WaterPress $\geq$ Low)	$\neg pred$ (WaterPress $\geq$ Low) $\wedge$ (WaterPress $\geq$ Low)	$\neg pred$ (WaterPress $\geq$ Low) $\wedge$ (WaterPress $\geq$ Low)
$\neg pred$ (Pressure=TooLow) $\wedge$ (Pressure=TooLow)	$\neg pred$ (Pressure=Permitted) $\wedge$ (Pressure=Permitted)	$\neg pred$ (Pressure=TooLow) $\wedge$ (Pressure=TooLow)	$\neg pred$ (Pressure=TooLow) $\wedge$ (Pressure=TooLow)	$\neg pred$ (Pressure=TooLow) $\wedge$ (Pressure=TooLow)
$\neg pred$ (Pressure=Permitted) $\wedge$ (Pressure=Permitted)	$\neg pred$ (Pressure=Permitted) $\wedge$ (Pressure=Permitted)	$\neg pred$ (Pressure=High) $\wedge$ (Pressure=High)	$\neg pred$ (Pressure=TooLow) $\wedge$ (Pressure=TooLow)	$\neg pred$ (Pressure=Permitted) $\wedge$ (Pressure=Permitted)
$\neg pred$ (Pressure=High) $\wedge$ (Pressure=High)	$\neg pred$ (Pressure=High) $\wedge$ (Pressure=High)	$\neg pred$ (Pressure=High) $\wedge$ (Pressure=High)	$\neg pred$ (Pressure=High) $\wedge$ (Pressure=High)	$\neg pred$ (Pressure=Permitted) $\wedge$ (Pressure=High)

Table 3.15: State transition table for *pressure* with the "pred" symbol.

STT: f-Pressure

	succ (WaterPress $\geq$ Low) $\wedge$ $\neg$ (WaterPress $\geq$ Low)	*succ* (WaterPress $\geq$ Low) $\wedge$ $\neg$ (WaterPress $\geq$ Low)	*succ* (WaterPress $\geq$ Low) $\wedge$ $\neg$ (WaterPress $\geq$ Low)	*succ* (WaterPress $\geq$ Low) $\wedge$ $\neg$ (WaterPress $\geq$ Low)
succ (Pressure=TooLow) $\wedge$ $\neg$ (Pressure=TooLow)	*succ* (Pressure=Permitted) $\wedge$ $\neg$ (Pressure=Permitted)	*succ* (Pressure=TooLow) $\wedge$ $\neg$ (Pressure=TooLow)	*succ* (Pressure=TooLow) $\wedge$ $\neg$ (Pressure=TooLow)	*succ* (Pressure=TooLow) $\wedge$ $\neg$ (Pressure=TooLow)
succ (Pressure=Permitted) $\wedge$ $\neg$ (Pressure=Permitted)	*succ* (Pressure=Permitted) $\wedge$ $\neg$ (Pressure=Permitted)	*succ* (Pressure=High) $\wedge$ $\neg$ (Pressure=High)	*succ* (Pressure=TooLow) $\wedge$ $\neg$ (Pressure=TooLow)	*succ* (Pressure=Permitted) $\wedge$ $\neg$ (Pressure=Permitted)
succ (Pressure=High) $\wedge$ $\neg$ (Pressure=High)	*succ* (Pressure=High) $\wedge$ $\neg$ (Pressure=High)	*succ* (Pressure=High) $\wedge$ $\neg$ (Pressure=High)	*succ* (Pressure=High) $\wedge$ $\neg$ (Pressure=High)	*succ* (Pressure=Permitted) $\wedge$ $\neg$ (Pressure=High)

Table 3.16: State transition table for *pressure* with the "succ" symbol.

Chapter 4

Tabular Expressions Composition

4.1 Introduction

The problem of composing a relation R from its parts R_α seems to be an open research problem [26]. One of the biggest advantages of adopting tabular expressions is to be able to easily specify a complex relation from its parts. In the context of tabular expressions the composition[1] of a relation R from its parts R_α is important for many reasons. For instance, it is very important to compose and decompose tables in a modular way. Therefore, we need to know what is the relation, and what are its parts. Also for large systems, the big table used to represent the system's behaviour could be decomposed into smaller tables that are more readable. This composition/decomposition is needed at different levels of the project life cycle.

Regarding the problem of the composition of tabular expression, we discuss the following approaches. In [26, 29, 28], Janicki et al. considered the composition of the cells of a table representing a relation or a function. Deharnais et al. introduced relational operators

[1] As Janicki pointed to it in [24], we also mean by composition "the act of putting together" and not the mathematical composition.

for the composition of relational scenarios presented by tabular expressions [9]. In [31], Kahl proposed a vertical and horizontal composition of tables. Mohrenschildt depicted the composition of tables representing mathematical functions [35]. In the following sections we describe each one of these approaches respectively. Section 4.6 is dedicated to discuss the advantages and disadvantages of these approaches.

4.2 Cells composition of relation/function

The relation R defining a tabular expression may be complex, but it can be built from a collection of relations R_α, where each R_α is easily specified. These relations are sometimes easily defined in some cells of the table. In other cases, a cell may refer to another table. In tabular expressions, C_T is a table composition rule which states how the global relation/function is built from the local ones. The shape of C_T depends on the kind of table used [26, 29, 28]. In the following, we discuss several relations and show how they could be composed from smaller and simpler relations. The examples are borrowed from [28].

The function f shown in Figure 3.1 and represented by the normal table drawn in Figure 3.3, is the composition of its local representations $f_{i,j}$, with $i = 1, 2, 3$, and $j = 1, 2$. For instance, $f_{3,2} : (-\infty, 0) \times (-\infty, 10) \longrightarrow$ Reals, and $f_{3,2}(x, y) = x - y$ for $(x, y) \in$ dom$(f_{3,2})$.

$$f = \bigcup_{i \in \{1,2,3\} \wedge j \in \{1,2\}} f_{ij}.$$

Let us consider the function g defined as:

	$x+y$	$x-y$	$y-x$
$y \geq 0$	$x < 0$	$0 \leq x < y$	$x \geq y$
$y < 0$	$x < y$	$y \leq x < 0$	$x \geq 0$

Figure 4.1: The function g defined by an inverted table

$$g(x,y) = \begin{cases} x+y \text{ if } (x<0 \wedge y \geq 0) \vee (x<y \wedge y<0) \\ x-y \text{ if } (0 \leq x<y \wedge y \geq 0) \vee (y \leq x \wedge y<0) \\ y-x \text{ if } (x \geq y \wedge y \geq 0) \vee (x \geq 0 \wedge y<0) \end{cases}$$

The above function g is represented by an inverted table as shown in Figure 4.1. It is the union of its local representations.

$$g = \bigcup_{i \in \{1,2,3\} \wedge j \in \{1,2\}} g_{ij}.$$

Now, let us consider the relation G defined as:

$$(x_1,x_2)G(y_1,y_2,y_3) \Longleftrightarrow \begin{cases} (y_1 = x_1 + x_2) \wedge (y_2 x_1 - x_2 = y_2^2) \\ \quad \wedge (y_3 + x_1 x_2 = |y_3|^3) \quad \text{if} \quad x_2 \leq 0 \\ (y_1 = x_1 - x_2) \wedge (x_1 + x_2 y_2 = |y^2|) \\ \quad \wedge (y_3 = x_1) \quad \text{if} \quad x_2 > 0 \end{cases}$$

This relation G represents a vector table as shown in Figure 4.2. The relation G is the composition of its local representations $G_{i,j}$, with $i = 1,2$, and $j = 1,2,3$. Relations $G_{1,1}$

	$x_2 \leq 0$	$x_2 > 0$
$y_1 =$	$x_1 - x_2$	$x_1 - x_2$
$y_2 \mid$	$y_2 x_1 + x_2 = y_2^2$	$x_1 + x_2 y_2 = y_2$
$y_3 \mid$	$y_3 + x_1 x_2 = y_3^3$	$y_3 = x_1$

Figure 4.2: The function G defined by a vector table

and $G_{2,1}$ are functions (this is indicated by the "=" symbol used after the variable y_1 in the left header). Relations $G_{i,2}$ and $G_{i,3}$ have y_2 and y_3 as their output variables (this is pointed by the symbol "|" used after the variables y_2 and y_3 in the left header). However, in this case the relation G is not the union of its local representations $G_{i,j}$. Instead;

$$G = \bigotimes_{j=1}^{3} \bigcup_{i=1}^{2} G_{ij},$$

where $\bigotimes$ is a generalisation of the intersection operator, and the join operator from relational data base theory [28].

The decision table representing the function φ is shown in Figure 4.3. The function φ is the composition of the local representations $\varphi_{i,j}$, with $i = 1, \ldots, 5$, and $j = 1, 2, 3$. For example, $\varphi_{3,2}$(*cloudy*) = *go to the beach*. The function φ it is not the union of its local representations $\varphi_{i,j}$. Instead,

$$\varphi = \bigcup_{i=1}^{5} \bigotimes_{j=1}^{2} \varphi_{ij}.$$

The function h presents the following generalized decision table.

$$h(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{if} \quad x_1 x_2 < 20 \wedge x_1/x_2 > 30 \\ x_1 - x_2 & \text{if} \quad x_1 x_2 \geq 20 \wedge x_1/x_2 < 30 \\ x_1 x_2 & \text{if} \quad x_1/x_2 = 30 \end{cases}$$

	go sailing	gtb	gtb	play bridge	garden
Temperature $\in$ {*hot, cool*}	*	*	*hot*	*	*cool*
Weather $\in$ {*sunny, cloudy, rainy*}	*sunny* $\vee$ *cloudy*	*sunny*	*cloudy*	*rainy*	*cloudy*
Windy $\in$ {*true, false*}	*tue*	*false*	*false*	*	*false*

* = don't care, gtb = go to the beach

Figure 4.3: The function φ defined by a table

	$x_1 + x_2$	$x_1 - x_2$	$x_1 x_2$
$x_1 x_2$	$\# < 20$	$\# \geq 20$	*true*
$x_1 \div x_2$	$\# > 30$	$\# < 30$	$\# = 30$

Figure 4.4: The function h defined by a (generalized decision) table

This function h can be represented by a generalized decision table as shown in Figure 4.4. The function h is the composition of its local representations $h_{i,j}$, with $i = 1, 2, 3$, and $j = 1, 2$. For example, $h_{2,1}(x_1, x_2) = x_1 - x_2$. Also for the function h, it is not the union of its local representations $h_{i,j}$,

$$h = \bigcup_{i=1}^{3} \bigcap_{j=1}^{2} h_{ij}.$$

In Appendix A, we show how we extend Janicki's work to compose tables instead of composing cells of one table. Some examples of composition of normal tables, inverted

tables, vector tables, and decision tables are provided as well.

4.3 Table composition of relational scenarios

In [9], tabular expressions are adopted to represent relational scenarios. A formal relational scenario is a triple (T, R_e, R_s) where T is a state space, R_e is the relation of the environment, R_s is the relation of the system. R_e and R_s are two disjoint relations defined on T. The tables are composed when the represented scenarios are consistent. Otherwise, the source of inconsistency is detected. Two scenarios are consistent when their demonic meet (discussed in the following) is defined. To compose two tables representing two scenarios (T, R_{e1}, R_{s1}) and (T, R_{e2}, R_{s2}), the resulting table will have the union of the relations of the environment, and the demonic meet of the relations of the system $(T, R_{e1} \cup R_{e2}, R_{s1} \sqcap R_{s2})$. The demonic meet of two relations P and Q is defined when:

$$dom(Q) \cap dom(R) = dom(Q \cap R) \tag{4.1}$$

Condition 4.1 means that any element in the common domain of Q and R, it must have at least one common image. When defined, the demonic meet of Q and R denoted by $Q \sqcap R$ is:

$$Q \sqcap R = (Q \cap R) \cup (\overline{R\top} \cap Q) \cup (\overline{Q\top} \cap R), \tag{4.2}$$

where $\top$ is the universal relation.

For instance, if P and Q are partial specifications of the same program, their demonic meet represents the combination of specifications. A program satisfies $Q \sqcap R$ if and only if it satisfies P and Q. To illustrate it with an example, let $Q \triangleq \{(0,3),(0,4),(1,5)\}$ and $R \triangleq \{(0,3),(0,6),(2,5)\}$. When their demonic meet is defined, it means they agree on the

actions to be carried from their common domain. In this case, the common domain for P and Q is $\{0\}$, and they agree on at least one image which is $\{3\}$. In this case, the demonic meet of Q and R is:

$$Q \sqcap R = \{(0,3),(1,5),(2,5)\}$$

However, if we slightly modify the previous example and take $Q = \{(0,3),(0,4),(1,5)\}$ and $R = \{(0,5),(0,6),(2,5)\}$, then the demonic meet is not defined since Condition 4.1 is not satisfied. In fact, for their common domain $\{0\}$, they have totally different images. We say that Q and R are inconsistent. In the following we give an example borrowed from [9] to illustrate scenarios composition represented by tabular expressions. The example is about a library system, and is described by two scenarios. The first one illustrates books checkout. The second one verifies the number of books that the user can borrow from the library. The two scenarios are informally discussed in the following.

Checkout scenario:

"The reader comes in. The system is in the initial state of the readerserv menu. The user enters the name of the reader. If the system does not know this name, then the user either

1. switches to the registration menu or
2. goes back to the initial state of the readerserv menu (and abandons the operation) or
3. reenters the name correctly.

If the system knows the reader's name, it asks whether the transaction is a checkout or a document return. The user may choose to return to the initial state of the readerserv menu (abandoning the operation) or choose the checkout option. In the latter case, for each document that the reader wants to borrow, the system displays the list of books already loaned to the reader and asks for the code of the new document; the user then enters the code of the borrowed document, the system adds the document to the set of documents

borrowed by the reader and also registers who the document is loaned to (so that it is possible to obtain the name of the borrower from a description of the document). The user finally returns the system to the initial state of the readerserv menu".

The following structures are needed to formalise the relation of the environment of the checkout scenario ($Checkout_e$), and the relation of the system of the checkout scenario ($Checkout_s$):

- A is a set of symbols. A^+ depict nonempty finite sequences of elements of A, and λ is the the empty sequence.
- C is a set of commands. In this example, commands start with the symbol '@'.
- *Readers* is the set of readers.
- r depicts the reader's name.
- *Documents* is the set of documents
- *Loanedto: Documents* $\rightarrow$ *Readers* is a partial function from a document to its borrower.
- *Has: Readers* $\rightarrow$ $\wp$*(Documents)* is a function mapping a reader to the documents he is borrowing.
- i is an input variable from the user to the system.
- o is an output variable from the system to the user. In Table 4.1, the abbreviated output messages and their meanings are represented.
- M is a variable indicating the menu's name.

Therefore, the space T_c of the checkout scenario is defined by the following variables:

Abbreviations	Meanings
name?	Enter the name of the reader
unk;name?	The name is unknown;enter the correct name
co,return?	Is the operation a checkout or a return?
address?	Enter the address
Has(r);doc?	<List of documents borrowed by r> Enter new document
Has(r);limit	<List of documents borrowed by r> Your limit is reached

Table 4.1: The abbreviations and meanings of the output messages

	o =name?	o=unk;name?	o=co,return?	o=Has(r);doc?	other
M=readerserv $\wedge i = \lambda$	$i' \in A^+$	$i' \in A^+$ $\vee i'$=@registration $\vee i'$=@readerserv	i'=@checkout $\vee i'$=@readerserv	false	false
M=checkout $\wedge i = \lambda$	false	false	false	$i' \in A^+$ $\vee i'$ =@readerserv	false
other	false	false	false	false	false

Figure 4.5: The relation of the environment of the checkout scenario ($Checkout_e$)

Readers, r, Documents, Loanedto, Has, i, o, M.

The tables corresponding to the relation of the environment of the checkout scenario ($Checkout_e$), and the relation of the system of the checkout scenario ($Checkout_s$) are represented in Figure 4.5 and Figure 4.6 respectively.

The second scenarios verifies the number of books that the user can borrow from the library. it is described in the following.

Limit-reached scenario:

"The reader comes in. The system is in the initial state of the readerserv menu. The user

	$\wedge$ o =name?	o=unk;name?	o=co;return?	o=Has(r);doc?	other
M=readerserv $\wedge\, i \in$ Readers	o=co;return? $\wedge$ r'=i $\wedge$ onlychange(i,o,r)	o=co;return? $\wedge$ r'=i $\wedge$ onlychange(i,o,r)	false	false	false
M=readerserv $\wedge\, i \in A^+ \wedge i \notin$ Readers	o=unk;name? $\wedge$ r'=i $\wedge$ onlychange(i,o,r)	r'=i $\wedge$ onlychange(i,o,r)	false	false	false
M=readerserv $\wedge$ i = @registration	false	M'=registration $\wedge$ o' = address? onlychange(M,i,o)	false	false	false
M=readerserv $\wedge$ i = @checkout	false	false	M'=checkout o'=Has(r);doc? onlychange(M,i,o)	false	false
M=readerserv $\wedge\, i$ = @readerserv	false	o'=name? onlychange(i,o)	o'=name? onlychange(i,o)	false	fasle
M=checkout $\wedge i \in A^+$	false	false	false	Has'=Has$\oplus(r \rightarrow (\mathrm{Has(r)} \cup \{i\}))$ Loandedto'=Loandedto $\cup(i \rightarrow r)$ onlychange (i,has,Loandeto)	false
M=checkout $\wedge i$ = @readerserv	false	false	false	M' = readerserv o' = name? onlychange (M,i,o)	false
other	false	false	false	false	false

Figure 4.6: The relation of the system of the checkout scenario ($Checkout_s$)

	o =name?	o=co;return?	o=Has(r);doc?	other
M=readerserv $\wedge i = \lambda$	$i \in A^+$	i'=@checkout	false	false
M=checkout $\wedge i = \lambda$	false	false	i'=@readerserv	false
other	false	false	false	false

Figure 4.7: The relation of the environment of the limit scenario ($Limit_e$)

inputs the name of the reader to the system. If the reader's name is known to the system, then the system asks whether the transaction is a checkout or the return of a document. The user chooses the checkout option. If the reader has reached his quota (limit number of books he is permitted to borrow), the system displays the list of books already loaned to the reader and indicates that the limit is reached. The user then chooses to return to the initial state of the readerserv menu".

For the limit-reached scenario, it has the same declarations as the checkout scenario with a new function *Limit: Readers* $\rightarrow$ *N*. The function *Limit* associates with each reader the limit of documents that she can borrow. The tables corresponding to the relation of the environment of the limit-reached scenario ($limit_e$), and the relation of the system of the limit-reached scenario ($limit_s$) are represented in Figure 4.7 and Figure 4.8 respectively.

The composition of the relations of the systems of the *checkout* scenario presented in Figure 4.6, and the *limit-reached* scenario presented in Figure 4.8 is obtained by computing the demonic meet of the two relations. It is represented in Figure 4.10.

	o =name?	o=co;return?	o=Has(r);doc?	other
M=readerserv $\wedge\, i \in$ Readers	o'=co;return? $\wedge$ r' = i $\wedge$ onlychange(i,o,r)	false	false	false
M=readerserv $\wedge\, i =$ @checkout $\wedge \lvert(Has(r)\rvert \geq$ Limit(r)	false	M'=checkout o'=Has(r);limit $\wedge$ onlychange(M,i,o)	false	false
M=checkout $\wedge\, i =$ @readerserv	false	false	M'=@readerserv o' = name? onlychange(M,i,o)	false
other	false	false	false	false

Figure 4.8: The relation of the system Limit ($Limit_s$)

	o =name?	o=unk;name?	o=co,return?	o=Has(r);doc?	o=Has(r);limit	other
M=readerserv $\wedge\, i = \lambda$	$i' \in A^+$	$i' \in A^+$ $\vee\, i'$=@registration $\vee\, i'$=@readerserv	i'=@checkout $\vee\, i'$=@readerserv	false	false	false
M=checkout $\wedge\, i = \lambda$	false	false	false	$i' \in A^+$ $\vee$ i'=@readerserv	i'=@readerserv	false
other	false	false	false	false	false	false

Figure 4.9: The relation of the environment of the checkout limit scenarios ($CheckoutLimit_e$)

	∧ o =name?	o=unk;name?	o=co;return?	o=Has(r);doc?	o=Has(r);limit?	other
M=readerserv ∧ $i \in$ Readers	o'=co;return? ∧ r' = i	o'=co;return? ∧r' = i	false	false	false	false
M=readerserv ∧ $i \in A^+$ ∧ $i \notin$ Readers	o=unk;name? ∧ r'=i ∧ onlychange(i,o,r)	r'=i ∧ onlychange(i,r)	false	false	false	false
M=readerserv ∧i = @registration	false	M' = registration ∧ o' = address? onlychange(M,i,o)	false	false	false	false
M=readerserv ∧ i = @checkout ∧$\|Has(r)\| \geq$ Limit(r)	false	false	M'=checkout ∧ o'=Has(r);doc? ∧ onlychange(M,i,o)	false	false	false
M=readerserv ∧ i = @checkout ∧$\|Has(r)\| \leq$ Limit(r)	false	false	M'=checkout ∧o' = Has(r);doc? ∧ onlychange(M,i,o)	false	fasle	false
M=readerserv ∧i = @readerserv	false	o'=name? onlychange (i,o)	o'=name? onlychange (i,o)	false	false	false
M=checkout ∧$i \in A^+$	false	false	false	Has'=Has$\oplus(r \rightarrow (\text{Has(r)} \cup \{i\}))$ ∧ Loanedto'=Loanedto $\cup(i \rightarrow r)$ ∧ onlychange (i,Has,Loanedto)	false	false
M=checkout ∧i =@readerserv	false	false	false	M'=readerserv ∧ o'=name? ∧ onlychange (M,i,o)	false	false
other	false	false	false	false	false	false

Figure 4.10: The relation of the system of the checkout limit scenarios ($Checkoutlimit_s$)

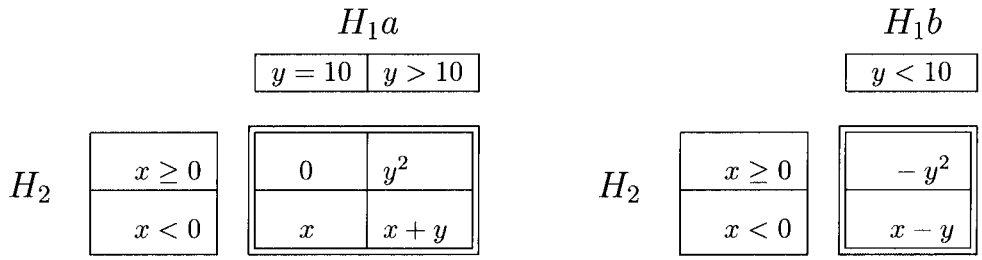


Figure 4.11: The tables $T_{f,a}$ and $T_{f,b}$

4.4 Horizontal and vertical table composition

In [31], Kahl claims that two-dimensional tables can be decomposed along any grid line. For instance, table T drawn in Figure 3.3 can be decomposed into the two tables $T_{f,a}$ and $T_{f,b}$ represented in Figure 4.11. In this case the decomposition of table T is done along header H_1 only. So, both resulting tables still have the same header H_2. To compose two tables of the same dimension, the table concatenation operator (|||) is used. For instance we have,

$$T = T_{f,a} ||| T_{f,b}.$$

The operator ||| is associative,

$$(t_1 ||| t_2) ||| t_3 = t_1 ||| (t_2 ||| t_3).$$

However, it is not commutative. Indeed, the table $T_{f,a}|||T_{f,b}$ is "graphically" different from the table $T_{f,b}|||T_{f,a}$.

The operator $\triangleright$ is used to concatenate a header h and an n-dimensional table t into an $(n+1)$ dimensional table $h \triangleright t$. In $h \triangleright t$, the header h will be the first dimension of the table, and the dimensions of t will be shifted in $h \triangleright t$. The table $T_{f,aa}$ represented in Figure

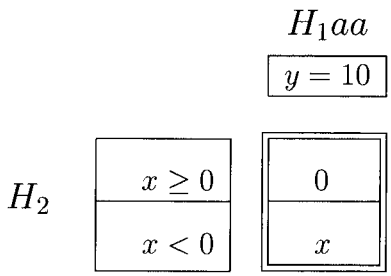


Figure 4.12: The table $T_{f,aa}$

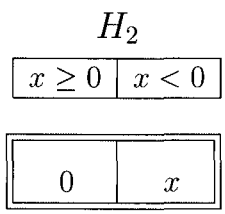


Figure 4.13: The one-dimensional table $T_{f,c}$

4.12 was decomposed into the header H_{1aa} and table $T_{f,c}$ drawn in Figure 4.13. The table $T_{f,c}$ was turned around since it has only one header and the first header is placed at the top. Now the operator $\triangleright$ is used to concatenate table $T_{f,c}$ with header H_{1aa} to obtain table $T_{f,aa}$.

$x \leq 0$	$x > 0$
$x \longmapsto -x$	$x \longmapsto 2x$

$m = A$	$m = B$
$x \longmapsto x + 1$	$x \longmapsto x + 2$

Figure 4.14: Table T_1 (left) and T_2 (right)

4.5 Table composition of mathematical functions

In [35], tabular expressions were chosen to represent conditional statements. The author proposed to slightly modify them in order to have a return value, and called them "state transformation tables". State transformation tables are quite similar to SCR event tables and SCR condition tables. A discussion on these tables was given in Section 2.3. The approach followed in [35] is based on an algebraic composition of state transformation tables using many sorted algebra. The proposed algorithms depend on the associativity of the functions the tables are built over. The mathematical function composition denoted by $\circ$ is adopted. For instance, the composition of the two functions $F := \{x_1 \longmapsto x_1 + x_2\}$ and $G := \{x_2 \longmapsto x_2 + 1\}$ results in $F \circ G = \{x_1 \longmapsto x_1 + x_2 + 1, x_2 \longmapsto x_2 + 1\}$ (where $\longmapsto$ denotes substitution). To compose tables, one has to check whether or not headers variables of the first table depend on the output variables of the second table. If headers variables of a table T do not depend on the output variables of a table T', the composition of the table $T = (H^1, H^2, ...H^k, G)$ and $T' = (H'^1, H'^2, ...H'^r, G')$ will result in a table $T'' = (H^1, H^2, ...H^k, H'^1, H'^2, ...H'^r, G'')$ such that its headers are the union of the headers of both table T and T', and its grid G'' is obtained by composing the two grids G and G'. As an example, the headers variables of table T_2 do not depend on the output variables of T_1 (see Figure 4.14). Therefore, for the table $T_2 \circ T_1$, its headers are the union of the headers of table T_2 and the headers of table T_1. Its grid is obtained by composing the grid of table T_2 to the grid of table T_1. The table composing T_2 and T_1 is given in Figure 4.15. If the

	$x < 0$	$x \geq 0$
$m = A$	$x \longmapsto -x + 1$	$x \longmapsto 2x + 1$
$m = B$	$x \longmapsto -x + 2$	$x \longmapsto 2x + 2$

Figure 4.15: Table $T_2 \circ T_1$

$m = A \wedge x + 1 < 0$	$x \longmapsto -x - 1$
$m = A \wedge x + 1 \geq 0$	$x \longmapsto 2x + 2$
$m = B \wedge x + 2 < 0$	$x \longmapsto -x - 2$
$m = B \wedge x + 2 \geq 0$	$x \longmapsto 2x + 4$

Figure 4.16: Table $T_1 \circ T_2$

headers variables of a table T depend on the output variables of a table T', the composition of the two tables T and T' will result in a table T'' with one header H''. H'' is obtained by composing each header cell of table T with each header cell of table T' with the respective grid entry of table T'. The number of grid entries is the product of the gird entries of the two tables T and T'. The new grid will be the composition of the grid entries of table T with the grid entries of tables T'.

For instance, $T_1 \circ T_2$ drawn in Figure 4.16 has one header combining the headers cells of tables T_1 and T_2, and the respective grid entry of table T_2. The grid of $T_1 \circ T_2$ is the composition of the grid entries of table T_1 with those of T_2.

4.6 Discussion

In this section we discuss the approaches previously presented, and depict the advantages and disadvantages of each one of them. Janicki et al. [26, 29, 28] introduced some operators

to compose table cells in order to obtain the relation or function represented by a tabular expression. C_T is the table composition rule which states how the global relation/function is built from the local ones. The shape of C_T depends on the kind of the tables used. However, instead of composing table cells, we were able to extend this work and compose tables. For the tables, we use the union operator to compose them. Then, for each kind of table, we use the appropriate operators to compose cells table as discussed in Section 4.2. The examples of table composition are presented in Appendix A. In [9], Desharnais et al. use relational algebra for table composition. Tabular expressions were adopted to represent relational scenarios. A formal relational scenario is presented by a triple (T, R_e, R_s) where T is a state space, R_e is the relation of the environment, and R_s is the relation of the system. To compose two tables representing two scenarios (T, R_{e1}, R_{s1}) and (T, R_{e2}, R_{s2}), the resulting table depicts the union of the relations of the environment, and the demonic meet of the relations of the system $(T, R_{e1} \cup R_{e2}, R_{s1} \sqcap R_{s2})$. However, what they proposed is restricted to relational scenarios only.

In [31], Kahl shows that the concatenation operator $|||$ used to concatenate tables is not commutative, and that by swapping columns, the produced tables are "graphically different". In fact, we do not agree on this since from a semantic point of view the composed tables are semantically equivalent. Therefore, such an operator should be commutative too. He also assumed that the concatenation operator $|||$ is used to concatenate tables of the same dimension. That worked in the example he adopted since he decomposed the table, and then recomposed it using the $|||$ operator. However, in general this is not true, since we are able to compose tables only if the variables have a common domain. Moreover, for the $\triangleright$ operator there is not always a need to concatenate a header and an n-dimensional table t into an (n+1) dimensional table $h \triangleright t$. In some cases, the new header to be added is part of or complement the other header. In fact, if we want to add header H_{1b} to table $T_{f,a}$ there is no need to add a new header to table $T_{f,a}$, since the values of header H_{1a} (first header of

table $T_{f,a}$) and header H_{1b} complement each other. As a result, table $T_{f,a}$ will remain with two dimensions.

In [35], the composition of two tables gives a table with one header. The number of grid entries is the product of the gird entries of the two tables to be composed. Hence, if the two tables are big, it will be disadvantageous to have a one-dimensional table since that resulting table will be of huge size. Moreover, the proposed approach in [35] is restricted to functions only, and is adopting the composition of functions, which cannot be applied to relations.

Under this approach, the composition of tabular expressions becomes the relational composition of matrices.

In the literature, there are other techniques that approached this tabular composition problem. In [45], Sekerinski tackled mainly the verification and refinement problem. However, he had some operators such as table conjunction, table disjunction, table negation, extending and contracting tables that could be seen as composition operators. We do have some similarities between these operators and ours that will be discussed with more details in the next chapter.

In [8], Desharnais et al. adopted an algebra of relations to define the semantics of tabular expressions. They used a similar notation to the APL programming language. In their work they considered 0-dimensional arrays called also scalars, 1-dimensional arrays called vectors, and 2-dimensional arrays called matrices. Headers are seen as vectors, and the grid as an n-dimensional array. That relational composition of matrices is a special case of an inner product.

In fact,

$$M; N = M \cup .; N$$

where ; is the relational composition operator.

For example,

$$(P \quad Q).\cup(R \quad S \quad T) = \begin{pmatrix} P \cup R & P \cup S & P \cup T \\ Q \cup R & Q \cup S & Q \cup T \end{pmatrix}$$

Chapter 5

Tabular Expressions Operators

5.1 Introduction

In this chapter we show how we improve the syntax and semantics of tabular expressions. Then, we present the operators that we introduced, and which are classified into unary operators, inner operators, and outer operators. Next, we define a partial order relation, and a refinement ordering relation on tabular expressions. Then, we propose an algebra of tabular expressions and finally, we present the verification process.

In the next section we show how we improve the syntax and semantics of tabular expressions. In Section 5.3, we present the tabular expressions operators that we introduced. In Section 5.4, we depict the partial order that we defined on tabular expressions. Section 5.5 is devoted to discuss the refinement relation specified on tabular expressions. In Section 5.6, we present our algebra of tabular expressions. Finally, section 5.7 is dedicated to present the verification process.

input variables	x,y: Reals
output variables	z: Reals
CCG	
P_T	$H_1 \wedge H_2$
r_T	G
Function name	f and z = f(x,y)
C_T	$\cup_{i=1}^{3} \cup_{j=1}^{2} f_{ij},$

Figure 5.1: Signature of the function f.

5.2 Improving the syntax and semantics of tabular expressions

As discussed in Section 3.2, Janicki et al. [28] defined formally a tabular expression T as:

$$T = (P_T, r_T, C_T, , CCG, H_1, ..., H_n, G, \Psi, IN, OUT).$$

The meaning (semantics) of a tabular expression T is given by a relation $R_T \subseteq IN \times OUT$, and is defined as:

$$R_T = C_T(R_\alpha).$$

The signature of a table is defined by the tuple:

$$Sign_T = (P_T, r_T, C_T, CCG),$$

and is presented by a two column table containing textual and graphical information. The signature of a table contains the input and output variables, and their types, the CCG, the table predicate rule, the table relation rule, the function name, and the table composition rule. For instance, the signature of the normal table drawn in Figure 3.10 is shown in Figure 5.1.

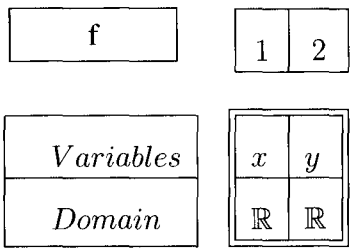


Figure 5.2: A declaration table for the function f.

In our work, for tabular expressions T_{Exp} we distinguish two main components the *declaration table* denoted by T_{dec}, and the *main table* itself denoted by T. Hence, we propose the following syntax for our tabular expressions.

$$T_{Exp} = (T_{dec}, T)$$

For declaration tables, we find out that it is more appropriate to represent them in a tabular way rather than having them depicted by a mixture of a graphical and textual way. In fact, the tabular notation increases their readability. Declaration tables could also be drawn at the left corner of the tabular expression. Hence, in our work, the declaration itself is a tabular expression containing the name of the function, variables indexes, variables and their domain. For instance, the table shown in Figure 5.2 is the declaration of the normal table drawn in Figure 3.10. From the declaration table, we are able to determine the kind of the table represented, Indeed, we also adopt the convention to represent the result values with a double border. From the kind of table, we could also infer the table composition rules which indicate how the global relation/function is built from local representations.

The main table depicts its content. It contains variables, constants, and operators applied on the constants and/or variables. Tables are viewed as a stack of atomic expressions and some operators that are applied on the atomic tabular expressions. Viewing tabular

expressions as stack of atomic expressions allows us to have build up of tables from the atomic ones. We distinguish two classes of tables which are *atomic tabular expressions* and *operator tabular expressions*. An *atomic* tabular expression is a tabular expression in which all the expressions are either variables or constants. The tables drawn in Figures 5.16 and 5.18 are examples of atomic tabular expressions. An *operator* tabular expression is a tabular expression in which the expressions are operators. The operators we propose are classified with into unary operators, inner operators, Kronecker operators, and outer operators. They are discussed in the following section.

5.3 Tabular expressions operators

In this section we discuss the different kinds of tabular expressions that we propose, and the operators that we propose. Before going that far, we would like to highlight two notions that we use in our work. Two tabular expressions have the same dimension, if they have the same number of headers. Two tabular expressions have the same size, if they have the same number of headers, the same number of cells in each header, and hence the same grid size. From that, we infer that two tables having the same size have the same dimension, but not vice versa.

5.3.1 Unary operator tabular expressions

In the following we define unary operator tabular expressions, their composition with tabular expressions, and we give examples to illustrate that.

(5.1) **Definition.** A *unary operator tabular expression* denoted T_{uop} is a tabular expression composed of cells that contains unary operators only.

(5.2) **Definition.** Given a tabular expression T and a unary operator tabular expression

	inc	*inc*	*inc*
inc	*inc*	*inc*	*inc*

inc = increment by 1.

Figure 5.3: Unary operator table T_{uop}.

	$x < 0$	$x = 0$	$x > 0$
$y = 0$	0	1	2

Figure 5.4: Table T_g.

T_{uop} with the same size as T, we define their composition $T_{uop}T$ by applying each unary operator in T_{uop} to the corresponding expression in the cell of T.

For instance, the application of the unary operator table T_{uop} shown in Figure 5.3 on the table T_g drawn in Figure 5.4 is given by the table presented in Figure 5.5.

In the case where T_{uop} has smaller size than T, we extract from T all the subtables having same size as T_{uop}. Then, we apply the unary operator table T_{uop} on each of the

	$x < 1$	$x = 1$	$x > 1$
$y = 1$	1	2	3

Figure 5.5: Table $T_{g_{op}}$

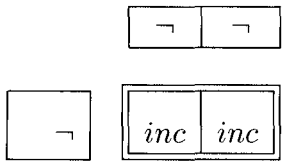


Figure 5.6: Unary operator table $T_{uop'}$.

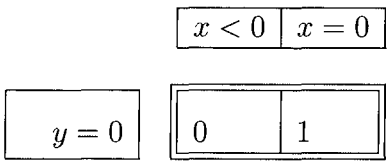


Figure 5.7: Table T_{g1}.

subtables of T. Hence, our result is a set of tabular expressions, and not a single table, which gives a more accurate information regarding the behaviour of the system. As another example, the unary operator table $T_{uop'}$ shown in Figure 5.6 has a smaller size than the table T_g drawn in Figure 5.4. Therefore, we start by extracting from table T_g all its subtables having same size as table $T_{uop'}$. The extraction gives three tables T_{g1}, T_{g2}, and T_{g3} shown in Figures 5.7, 5.8, and 5.9 respectively. Then, after applying the unary operator table $T_{uop'}$ depicted in Figure 5.6 on each subtable, the result is given in three tables shown in Figure 5.10, 5.11, and 5.12 respectively.

Of particular importance, we underline the *negation grid unary* operator that negates only grid cells, and not the header cells. It is denoted by $\neg_g$.

For instance, the application of the *negation grid unary* operator shown in Figure 5.13 on the table drawn in Figure 5.14 is given by the table presented in Figure 5.15.

	$x < 0$	$x = 0$
$y = 0$	0	1

Figure 5.8: Table T_{g2}.

	$x = 0$	$x > 0$
$y = 0$	1	2

Figure 5.9: Table T_{g3}.

	$\neg(x < 0)$	$\neg(x = 0)$
$\neg(y = 0)$	1	2

Figure 5.10: Table $T_{g1op'}$.

	$\neg(x < 0)$	$\neg(x > 0)$
$\neg(y = 0)$	1	3

Figure 5.11: Table $T_{g2op'}$.

	$\neg(x = 0)$	$\neg(x > 0)$
$\neg(y = 0)$	2	3

Figure 5.12: Table $T_{g3op'}$.

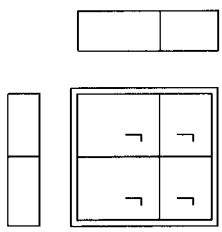


Figure 5.13: Negation grid operator table.

	$x = 0$	$x \neq 0$
$y = 1$	$True$	$False$
$y \neq 1$	$False$	$False$

Figure 5.14: Tabular representation of the negation of the relation R.

	$x = 0$	$x \neq 0$
$y = 1$	$False$	$True$
$y \neq 1$	$True$	$True$

Figure 5.15: Tabular representation of the negation of the relation R.

	y	y
x	y	y
x	x	x

Figure 5.16: Atomic table T_1.

	$=$	$>$
$\geq$	$-$	$*$
$<$	$*$	$+$

Figure 5.17: Operator table $_{inner}$.

5.3.2 Inner operators tabular expressions

In the sequel, we define inner operator tabular expressions, their composition with tabular expressions, and we provide examples to illustrate that.

(5.3) **Definition.** An *inner operator tabular expression* is a tabular expression in which all the expressions are binary operators.

	10	10
0	x	y
0	x	y

Figure 5.18: Atomic table T_2.

	$y = 10$	$y > 10$
$x \geq 0$	$y - x$	y^2
$x < 0$	x^2	$x + y$

Figure 5.19: Table $T_{1 .inner T_2}$.

(5.4) **Definition.** Given two tabular expressions T_1 and T_2 and an inner operator $._{inner}$ all having the same size, their composition $T_{1 ._{inner} T_2}$ is obtained by composing each cell from T_1, the corresponding cell from $._{inner}$, and T_2.

For example, the composition of the atomic table T_1, the inner operator table $._{inner}$, and the atomic table T_2 drawn in Figure 5.16, 5.17, and 5.18 respectively, gives the table $T_{1 .inner T_2}$ depicted in Figure 5.19.

The inner operator is also defined on two tables T_1 and T_2 where T_1 has smaller size than T_2. Given an inner operator tabular expression with same dimension as T_1, we can form a set of tabular expressions by computing the composition of T_1, the inner operator, and each sub tabular expression of T_2 that has the same dimension as T_1 (when computing the sub tabular expressions the order must be respected). For instance, table T_1 shown in Figure 5.16 has smaller size than table T_2' drawn in Figure 5.20. Hence, to compose the two tables with the inner operator presented in Figure 5.17, we start by extracting from table T_2' all its subtables having the same size as T_1. The tables extracted from table T_2' are shown in Figures 5.18, 5.21, and 5.22 respectively. The composition of the atomic table T_1, the inner operator table $._{inner}$, and each extracted table from T_2, gives three tables. The composed tables are shown in Figures 5.19, 5.23, and 5.24 respectively.

	10	10	10
0	x	y	y
0	x	y	2

Figure 5.20: Atomic table T_2'.

	10	10
0	x	y
0	x	2

Figure 5.21: Atomic table T_{22}.

	10	10
0	y	y
0	y	2

Figure 5.22: Atomic table T_{23}.

	$y = 10$	$y > 10$
$x \geq 0$	$y - x$	y^2
$x < 0$	x^2	$x + 2$

Figure 5.23: Table $T_{1.inner}T_{22}$.

	$y = 10$	$y > 10$
$x \geq 0$	0	y^2
$x < 0$	$x * y$	$x + 2$

Figure 5.24: Table $T_{1.inner}T_{23}$.

5.3.3 Kronecker operators tabular expressions

In the following, we define Kronecker operators tabular expressions and their composition with tabular expressions. Then, we give examples to illustrate that.

(5.5) **Definition.** A tabular Kronecker operator is a tabular expression in which all the expressions are binary operators such as $\cap, \cup, \wedge, *$, etc. Given two tabular expressions T_1 and T_2, and a Kronecker operator tabular expression $[K]$, their composition $T_1[K]T_2$ is the set of tabular expressions obtained by composing every subtable of size one from T_1 with the Kronecker operator and table T_2. A subtable of size one is obtained by a cell from each header with their corresponding grid cell.

For instance, the composition of tables $T_{K1}[K]T_{K2}$ shown in Figure 5.25, 5.26, and 5.27 respectively, gives the four tables drawn in Figures 5.28, 5.29, 5.30, and 5.31 respectively. Later in Section 5.4, we give an example of Kronecker composition where the binary operator is $\wedge$.

	$y \geq 10$	$y < 10$
$x \geq 0$	1	2
$x < 0$	3	4

Figure 5.25: Table T_{K1}.

	*	*
	*	*

Figure 5.26: Kronecker operator table $[K]$.

	$y \geq 10$	$y < 10$
$x \geq 0$	5	6
$x < 0$	7	8

Figure 5.27: Table T_{K2}.

	$y \geq 10$	$y < 10$
$x \geq 0$	5	6
$x < 0$	7	8

Figure 5.28: Table $T_{1K1[K]K2}$.

	$y \geq 10$	$y < 10$
$x \geq 0$	10	12
$x < 0$	14	16

Figure 5.29: Table $T_{2K1[K]K2}$.

	$y \geq 10$	$y < 10$
$x \geq 0$	15	18
$x < 0$	21	24

Figure 5.30: Table $T_{3K1[K]K2}$.

	$y \geq 10$	$y < 10$
$x \geq 0$	20	24
$x < 0$	28	32

Figure 5.31: Table $T_{4K1[K]K2}$.

$x < 0$	$x = 0$	$x > 0$
a	b	c

Figure 5.32: Table T_{j1}

The Kronecker operator is a generalisation of the join operator [1]. In [28], the join operator adopted by Janicki and Wassyng is defined as:

$$P \bowtie Q = \{(x, y) | x \in \Pi_{t \in J \cup L} D_t \wedge y \in \Pi_{t \in K \cup M} D_t \wedge (x|_J, y|_K) \in P \wedge (x|_L, y|_M) \in Q\}$$

where T is a set of indices, $\{D_t | t \in T\}$ is family of sets, J, K, L, M are subsets of T, and P, Q are the relations

$$P \subseteq \prod_{t \in J} D_t \times \prod_{t \in K} D_t, Q \subseteq \prod_{t \in L} D_t \times \prod_{t \in M} D_t.$$

The notation $\prod_{t \in J} D_t$ stands for the direct product of D_t, and $x|_J$ designates the projection of x on J.

For example let us consider the tables T_{j1} and T_{j2} and their natural join drawn in Figures 5.32, 5.33, and 5.34 respectively.

The Kronecker composition of the tables $T_{j1}\ [K]\ T_{j2}$ shown respectively in Figures 5.32, and 5.33 will result in three tables drawn respectively in Figures 5.35, 5.36, and 5.37.

The tables T'_{j1}, T'_{j2}, and T'_{j3}, drawn in Figures 5.35, 5.36, and 5.37 respectively is equivalent to the table $T_{j1} \bowtie T_{j2}$ given in Figure 5.34.

[1] The join operator is the same as the natural join database operator. It is denoted by R ⋈ S, where R and S are relations. The result of the natural join is the set of all combinations of tuples in R and S that are equal on their common domain.

	$x < 0$	$x = 0$
$y < 0$	d	e
$y = 0$	f	g
$y > 0$	h	i

Figure 5.33: Table T_{j2}.

H_1 \ H_2	$x < 0$	$x = 0$	$x > 0$
$y < 0$	$a \wedge d$	$b \wedge e$	c
$y = 0$	$a \wedge f$	$b \wedge g$	c
$y > 0$	$a \wedge h$	$b \wedge i$	c

Figure 5.34: Table $T_{j1} \bowtie T_{j2}$.

	$x < 0$	$(x < 0) \wedge (x = 0)$
$y < 0$	$a \wedge d$	$a \wedge e$
$y = 0$	$a \wedge f$	$a \wedge g$
$y > 0$	$a \wedge h$	$a \wedge i$

Figure 5.35: Table T'_{j1}.

	$(x < 0) \wedge (x = 0)$	$x = 0$
$y < 0$	$b \wedge d$	$b \wedge e$
$y = 0$	$b \wedge f$	$b \wedge g$
$y > 0$	$b \wedge h$	$b \wedge i$

Figure 5.36: Table T'_{j2}.

	$(x < 0) \wedge (x > 0)$	$(x = 0) \wedge (x > 0)$	$x > 0$
$y < 0$	$c \wedge d$	$c \wedge e$	c
$y = 0$	$c \wedge f$	$c \wedge g$	c
$y > 0$	$c \wedge h$	$c \wedge i$	c

Figure 5.37: Table T'_{j3}.

		H_2		
		$y = 10$	$y > 10$	$y < 10$
H_1	$x \geq 0$	$y - x$	y^2	$-y^2$
	$x < 0$	x^2	$x + y$	$x - y$

Figure 5.38: Table T_e

5.3.4 Outer operators tabular expressions

Outer operators are composed of extractors called also reducers, used to reduce the tables, and expanders used to expand the tables. In the following, we analyse the extractors and expanders, and we give some illustrative examples.

(5.6) **Definition.** A *tabular extractor operator* denoted by "\" allows us to extract part of the tabular expression. The header cells indexes are specified to determine the extracted table. It is defined as the follwing:

$$H_{1\backslash(m,\cdots,n)}, H_{2\backslash(p,\cdots,q)}, \cdots, H_{m\backslash(r,\cdots,s)}.$$

For instance, the application of the extractor operator $(H_{1\backslash(1,2)}, H_{2\backslash(2,3)})$ on the table T_e drawn in Figure 5.38 gives the extracted table given in Figure 5.39. From table T_e, we keep the first and the second cells from the first header, the second and third cells from the second header, and their corresponding grid cells.

Expander operators tabular expressions

We distinguish two kinds of expansion: a spatial expansion, and a dimensional expansion.

	$y > 10$	$y < 10$
$x \geq 0$	y^2	$-y^2$
$x < 0$	$x + y$	$x - y$

Figure 5.39: Extracted table T_{extr}.

(5.7) **Definition.** Given a table T, the spatial expansion operator denoted "Ⓢ" consists in adding expressions to the table, and hence will increase the size of its grid.

$$T \text{ Ⓢ } expression.$$

For example, we want to expand the table T_s drawn in Figure 5.40, by adding the expression $(x < 0)$ to the first header. The new space expanded table T_{sexp} is given in Figure 5.41. In the case where we do not have any information about the grid cells, we fill the empty cells with $False$.

(5.8) **Definition.** The dimensional expansion denoted by "Ⓢ", consists in adding new headers, and this may or may not increase the size of the grid.

$$T \text{ Ⓢ} expression.$$

For example, we expand the table T_d drawn in Figure 5.42 by adding a new dimension to it. The new header consists in one cell of value $(y = 10)$, and the new table T_{ddexp} is given in Figure 5.43. In this case, we did not increase the size of the grid. However, if we want to add to the table T_{ddexp} drawn in Figure 5.43 a new header containing a header cell with the value $(y > 10)$, and two grid cells with respective values y^2, and $x + y$, then the size of

	$y > 10$	$y < 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$

Figure 5.40: Table T_s.

	$y > 10$	$y < 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	$False$	$False$	$False$

Figure 5.41: Table T_{Sexp}.

the grid will increase. The new table is shown in Figure 5.44.

5.4 Partial order on tabular expressions

The motivation for introducing a partial order into tables is to formalize some tabular expressions aspects such as simplification of a tabular expression. In the following we consider predicate expressions tables.

(5.9) **Definition.** A tabular expression is said to be $false$, if each grid cell is $false$ or one of its corresponding header cells is $false$.

(5.10) **Definition.** Two tables T_1 and T_2 are *semantically equivalent* that we denote by $T_1 \sim T_2$, if they represent the same relations.

(5.11) **Definition.** $\tilde{T} = \{S | S \sim T\}$

$x \geq 0$	0
$x < 0$	x

Figure 5.42: Table T_d.

	$y = 10$
$x \geq 0$	0
$x < 0$	x

Figure 5.43: Table T_{ddexp}

	$y = 10$	$y > 10$
$x \geq 0$	0	y^2
$x < 0$	x	$x + y$

Figure 5.44: Table $T_{d'dexp}$

$x < 0$	$x = 0$	$x > 0$
$-x$	x	x

Figure 5.45: Table S_1

$x < 0$	$x \geq 0$
$-x$	x

Figure 5.46: Table S_2

Given a table T, let $\tilde{T}$ be the class of tables semantically equivalent to it. Then, $\sim$ is an equivalence relation. That is, reflexive, transitive, and symmetric.

(5.12) **Definition.** Size-of-$\tilde{T}$= Size-of-S, where $S \in \tilde{T}$ with S irreducible.

A table is irreducible if its size cannot be reduced. For example, tables S_1 and S_2 drawn in Figure 5.45 and 5.46 respectivelyare semantically equivalent. However, table S_1 is reducible, it could be reduced to table S_2 which is irreducible.

(5.13) **Definition.** $S_1[\leq]S_2 \iff S_1[K]\neg_g S_2$ is False, where S_1 and S_2 are proper tables, $[K]$ is the Kronecker operator, and $\neg_g$ is the negation grid operator.

A table is *proper* if for each header, all header cells are pairwise disjoint [38]. Informally, $S_1[\leq]S_2$ means that S_1 is part of S_2.

Theorem 5.4.1 $[\leq]$ *is a partial order relation between tabular expressions.*

PROOF. To prove that $[\leq]$ is a partial order, we have to show that it is reflexive, symmetric, and transitive.

- To prove that $[\leq]$ is reflexive, we have to show that $\forall T.\ T[\leq]T$.

  Let $T_1(h_1, h_2, \cdots, h_n, g_n)$ be an extracted table from $T(H_1, H_2, \cdots, H_n, G_n)$, where $h_1, h_2, \cdots, h_n$ are the headers of T_1 and g_n its grid, and $H_1, H_2, \cdots, H_n$ are the headers of T and G_n its grid. Let $T'(H'_1, H'_2, \cdots, H'_n, \neg G'_n)$ be the complement of the table T, where $H'_1, H'_2, \cdots, H'_n$ are the headers of T' and $\neg G'_n$ its grid.

  Since T is proper, we have the following cases:

  if $h_1 = H'_1, ...h_n = H'_n, \Rightarrow g_n = G'_n \Rightarrow g_n \wedge G'_n$ is $False$

  else $(h_1 \wedge h_2, ... \wedge h_n \wedge g_m) \wedge (H'_1 \wedge H'_2, ... \wedge H'_n, \wedge \neg G'_m)$ is $False$

  Thus, $T_1[K]\neg_g T$ is $False$, hence $T[\leq]T$.

- To prove that $[\leq]$ is antisymmetric, we have to show that $T_1[\leq]T_2 \ \wedge \ T_2[\leq]T_1 \Rightarrow T_1 = T_2$.

  Let R_1 be the relation representing the table $T_1(H_1, ..H_m, G_m)$, R_2 the relation representing the table $T_2(H'_1, ..H'_n, G'_m)$, and its complement

  $\neg T_2(H'_1, ..H'_n, \neg G'_n)$. Let $(x_1, x_2, ..., x_q)$ be a tuple in R_1. Assume by contradiction that $(x_1, x_2, ..., x_q)$ is not in R_2.

  Since T_2 is proper and $T_1[\leq]T_2$, there exists a unique $(H'_1, ..., H'_n)$ such that $(x_1, ..., x_q)$ is in $(H'_1, ..., H'_n)$.

  Let G'_m be the grid cell associated with $(H'_1, ..., H'_n)$. Since $(x_1, ..., x_q)$ is not in R_2, then $(x_1, ..., x_q)$ is in $\neg G'_m$. Thus $G_m \wedge \neg G'_n$ is $False$. This contradicts our assumption that $T_1[\leq]T_2$. Therefore, $R_1 \subseteq R_2$.

By exchanging the indices 1 and 2, we obtain $R_2 \subseteq R_1$. Thus, $R_1 = R_2$. Hence, $T_1 = T_2$.

- To prove that $[\leq]$ is transitive, we have to show that $T_1[\leq]T_2 \wedge T_2[\leq]T_3 \Rightarrow T_1[\leq]T_3$.

  Let $T_1(H_1, ..H_m, G_m)$, $T_2(H'_1, ..H'_n, G'_n)$, and $T_3(H''_1, ..H''_p, G''_p)$,

  if $(H_1 \wedge H_2, .. \wedge H_m) \wedge (H''_1 \wedge H''_2, .. \wedge H''_p)$ is $False$ then $T_1[K]\neg_g T_3$ is $False$, hence $T_1[\leq]T_3$

  else if $G_m \wedge \neg G''_p$ is $False$ then $T_1[K]\neg_g T_3$ is $False$, hence $T_1[\leq]T_3$,

  else let us suppose by contradiction that there exists $(x_1, ..., x_m)$ that is in $(H_1, ...H_m, G_m)$ but it is not in $(H''_1, ...H''_p, G''_p)$.

  By hypothesis we have, $T_1[\leq]T_2$, therefore there exists a unique $(H'_1, ..., H'_q)$ such that $(x_1, ..., x_m)$ is in $H'_1 \wedge H'_2 ... \wedge H'_q \wedge G'_q$. Given that $T_2[\leq]T_3$, the tuple $(x_1, ..., x_m)$ is in a unique $(L''_1, ...L''_p, R''_p)$. Since $(x_1, ..., x_m)$ is in $(H''_1 \wedge H''_2 ... \wedge H''_p, R''_p)$, then $(L''_1, ...L''_p, G''_p) = (H''_1, ..H''_p, R''_p)$ (by uniqueness) Therefore, $(x_1, ..., x_m)$ is not in $G_m \wedge \neg G''_p$, which contradicts our assumption. Hence, $T_1[\leq]T_3$.

□

Example 5.4.1 *To make our discussion simple, we take a predicate expression table where the elements of the headers and the grid are predicate expressions.*

For instance, to show that table T_p drawn in Figure 5.47 is $[\leq]$ than table T_q shown in Figure 5.48, we have to show that $T_1[K]\neg_g T_2$ is False.

The application of the negation grid operator shown in Figure 5.49 on Table T_q drawn in Figure 5.48 gives the Table T'_q drawn in Figure 5.50. Now to apply the Kronecker operator which is the binary operator $\wedge$ in this example, we have to extract from T_p all the subtables of size 1. So in this case, we will have 6 subtables. The first extracted table is given in

		H_2	
		$w < 0$	$w = 0$
H_1	$x = 3$	$y = 5$	$x + y = w$
	$x < 3$	$y > 7$	$y - x = 6$
	$x > 3$	$y^2 = 4$	$y^2 = 4$

Figure 5.47: A predicate expression table T_p.

		H_2		
		$w < 0$	$w = 0$	$w > 0$
H_1	$x = 3$	$y = 5$	$x + y = w$	$x + y = z$
	$x < 3$	$y > 7$	$y - x = 6$	$y - y = z$
	$x > 3$	$y^2 = 4$	$y^2 = 4$	$z = y$

Figure 5.48: A predicate expression table T_q.

figure 5.51. The application of the Kronecker operator on tables T_{p1} and $\neg_g T_q$ will result in the table drawn in Figure 5.52.

The simplified table $T_{p1}[K]\neg_g T_q$ drawn in Figure 5.53 is a false table. The application of the Kronecker operator between each subtable of T_p with $\neg T_q$ results in a false table.

(5.14) **Definition.**

$T_1[\leq]_D T_2$ iff $T_1[\leq]T_2$, size-of-T_1 $\leq$ size-of-T_2, and the disjunction of the respective headers cells of the two tables is the same.

Theorem 5.4.2 $[\leq]_D$ *is a partial order relation between tabular expressions.*

The proof goes along the same lines as the one of theorem 5.4.1.

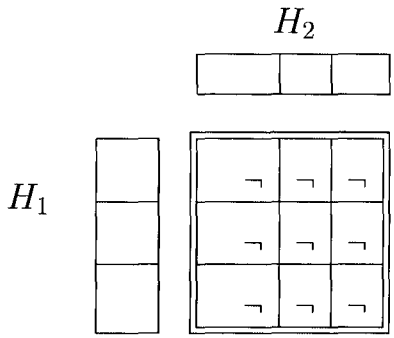


Figure 5.49: Negation operator table.

		H_2		
		$w < 0$	$w = 0$	$w > 0$
H_1	$x = 3$	$\neg(y = 5)$	$\neg(x + y = w)$	$\neg(x + y = z)$
	$x < 3$	$\neg(y > 7)$	$\neg(y - x = 6)$	$\neg(y - y = z)$
	$x > 3$	$\neg(y^2 = 4)$	$\neg(y^2 = 4)$	$\neg(z = y)$

Figure 5.50: The negation of the predicate expression table T'_q

	H_2
	$w < 0$
H_1	
$x = 3$	$(y = 5)$

Figure 5.51: Table T_{p1}

H_1 \ H_2	$(w < 0) \wedge (w < 0)$	$(w < 0) \wedge (w = 0)$	$(w < 0) \wedge (w > 0)$
$(x = 3) \wedge (x = 3)$	$(y = 5) \wedge \neg(y = 5)$	$(y = 5) \wedge \neg(x + y = w)$	$(y = 5) \wedge \neg(x + y = z)$
$(x = 3) \wedge (x < 3)$	$(y = 5) \wedge \neg(y > 7)$	$(y = 5) \wedge \neg(y - x = 6)$	$(y = 5) \wedge \neg(y - y = z)$
$(x = 3) \wedge (x > 3)$	$(y = 5) \wedge \neg(y^2 = 4)$	$(y = 5) \wedge \neg(y^2 = 4)$	$(y = 5) \wedge \neg(z = y)$

Figure 5.52: Table $T_{p1}[K]\neg_g T_q$.

H_1 \ H_2	$w < 0$	$False$	$False$
$x = 3$	$False$	$\neg(y = 5) \wedge (x + y = w)$	$\neg(y = 5) \wedge (x + y = z)$
$False$	$\neg(y = 5) \wedge (y > 7)$	$\neg(y = 5) \wedge (y - x = 6)$	$\neg(y = 5) \wedge (y - y = z)$
$False$	$\neg(y = 5) \wedge (y^2 = 4)$	$\neg(y = 5) \wedge (y^2 = 4)$	$\neg(y = 5) \wedge (z = y)$

Figure 5.53: Simplified table $T_{p1}[K]\neg_g T_q$.

(5.15) **Definition.** A function $\mathcal{S}$ from tabular expressions to tabular expressions is said to be a simplification function if $\mathcal{S}(T)[\leq]_D T$ and $\mathcal{S}$ is strictly monotonic with respect to $[\leq]_D$.

It is easy to see the following sequence has a least element, which is the fix point of $\mathcal{S}^n(T_n)$.

$$\begin{aligned} T_0 &= T \\ T_{n+1} &= \mathcal{S}(T_n). \end{aligned}$$

5.5 Tabular expressions refinement

For tabular expressions, the refinement relation $\subseteq$ is defined as:

(5.16) **Definition.** $TE_1 \subseteq TE_2 \Longleftrightarrow TE_1[\leq]TE_2$

A tabular expression refines another tabular expression if it is part of it. A table refines another means that it simplifies it. The refinement relation is a partial order, and that comes from the fact that is $[\leq]$ is a partial order.

Lemma 5.5.1 $\forall T \in TE.T_{False} \subseteq T$, *where* T_{False} *is a table in which all the values are* $False$.

PROOF.

$T_{False} \subseteq T$

$\Longleftrightarrow <$ Definition 16 $>$

$T_{False}[\leq]T$

$\Longleftrightarrow <$ Definition 13 $>$

$T_{False}[K]\neg_g T \Leftrightarrow False$

$\Longleftrightarrow$ True

□

Lemma 5.5.2 $\forall T \in TE.T \subseteq T_{Neutral}$, *where* $T_{Neutral}$ *is a table in which all the values are* $True$

PROOF.

$T \subseteq T_{Neutral}$

$\Longleftrightarrow <$ Definition 16 $>$

$T[\leq]T_{Neutral}$

$\Longleftrightarrow <$ Definition 13 $>$

$T[K]\neg_g T_{Neutral} \Leftrightarrow False$

$\Longleftrightarrow <$ Application of the negation operator $>$

$T[K]T_{False} \Leftrightarrow False$

$\Longleftrightarrow$ True

□

	$y > 10$	$y < 10$
$x \geq 0$	0	y^2

Figure 5.54: Table T_{s1}

	$y > 10$	$y = 10$
$x \geq 0$	0	$-y^2$

Figure 5.55: Table T_{s2}

(5.17) **Proposition.** *Given a table T, let $\mathcal{P}(T)$ be the power set of the table T containing all the sub-tables that refine it, and $\subseteq$ be the refinement relation. The structure $(\mathcal{P}(T), \subseteq)$ is a lattice. The greatest element of the lattice is the $neutral$ table ($T_{Neutral}$), and its least element is the $False$ table (T_{False}).*

In our lattice, we have the following properties:

$$T_1 \subseteq T_2 \quad \text{iff} \quad T_1 \text{ is extracted from } \quad T_2$$

$$T_1 \subseteq T_2 \quad \text{iff} \quad T_2 \text{ is expanded from } \quad T_1$$

The extractor and expanded operators were discussed in Subsection .

Example 5.5.1 *For instance the empty table, and the tables drawn in Figures 5.54, 5.55, 5.56, 5.57, 5.58, 5.59, respectively, refine the table T_s presented in Figure 5.40.*

	$y < 10$	$y = 10$
$x \geq 0$	y^2	$-y^2$

Figure 5.56: Table T_{s3}

	$y > 10$
$x \geq 0$	0

Figure 5.57: Table T_{s4}

	$y < 10$
$x \geq 0$	y^2

Figure 5.58: Table T_{s5}

	$y = 10$
$x \geq 0$	$-y^2$

Figure 5.59: Table T_{s6}

5.6 Algebra of tabular expressions

In the following we define an algebra of tabular expressions. We consider predicate expressions tables.

(5.18) **Definition.** An algebra of tabular expressions is a structure

$$(\widetilde{TE}, \widetilde{T}_{inner}, \widetilde{T}[K], \widetilde{T}_{False}, \widetilde{T}_{Neutral})$$

where $\widetilde{TE}$ is a class of tabular expressions which are semantically equivalent, $\widetilde{T}_{inner}$ is a class of inner operator tabular expressions, $\widetilde{T}[K]$ is a class of Kronecker operator tabular expressions, $\widetilde{T}_{False}$ is a class of $False$ tabular expressions, and $\widetilde{T}_{Neutral}$ is a class of $Neutral$ tabular expressions.

$\forall T_1, T_2,$ and $T_3 \in \widetilde{TE}$, our algebra of tabular expressions satisfies the following axioms:

(5.1) $(T_{1.inner}T_2)_{.inner}T_3 = T_{1.inner}(T_{2.inner}T_3)$

(5.2) $T_{1.inner}T_{False} = T_1$

(5.3) $T_{1.inner}T_2 = T_{2.inner}T_1$

(5.4) $(T_1[K]T_2)[K]T_3 = T_1[K](T_2[K]T_3)$

(5.5) $T_1[K]T_{Neutral} = T_1$

(5.6) $T_1[K](T_{2.inner}T_3) = (T_1[K]T_2)_{.inner}(T_1[K]T_3)$

(5.7) $(T_{1.inner}T_2)[K]T_3 = (T_1[K]T_2)_{.inner}(T_2[K]T_3)$

(5.8) $T_1[K]T_{False} = T_{False}[K]T_1 = T_{False}$

Theorem 5.6.1 *The structure* $(\widetilde{TE}, \widetilde{T}_{inner}, \widetilde{T}[K], \widetilde{T}_{False})$ *is a semiring.*

PROOF. $(\widetilde{TE}, \widetilde{T}_{inner})$ is a commutative monoid with identity element T_{False}. It satisfies axioms 5.1, 5.2, and 5.3. $(\widetilde{TE}, \widetilde{T}[K])$ is a monoid with identity element $T_{Neutral}$. It satisfies axioms 5.4, and 5.5. $\widetilde{T}_{inner}$ distributes over $T[K]$. It satisfies axioms 5.6, and 5.7. $\widetilde{T}_{False}$ annihilates $\widetilde{TE}$, with respect to $T[K]$. It satisfies axioms 5.8. Hence the structure $(\widetilde{TE}, \widetilde{T}_{inner}, \widetilde{T}[K], \widetilde{T}_{False})$ is a semiring.

5.7 Consistency and completeness

For requirements specifications, it is fundamental to check for properties such consistency and completeness. With their formal structure, tabular expressions are very convenient to check for consistency and completeness [28, 51]. The disjointness of the headers of the table ensures that the specification is consistent. Therefore, for each header H^k of size n, the following property should be satisfied:

$$\wedge(i, j | 1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j : H^k[i] \wedge H^k[j]) \Longleftrightarrow False$$

Tabular expressions are very helpful to verify partial completeness with the domain coverage theorem. The complete input domain coverage ensures that we have specified responses to every input combination. Also, the domain coverage condition also can be easily verified. It consists in verifying the following property:

$$\vee(i | 1 \leq i \leq n : H^k[i]) \Longleftrightarrow True$$

The exact verification formulae depends on the type of tabular expression, but is rather straightforward in each case [28, 51]. Also for our tables, we propose to adopt the disjoint-

ness of the headers to check for consistency, and the domain coverage property to check for completeness. We do not require our tables to be space complete. In fact, while gathering the requirements, each user has a partial view of the system, and hence the tables will not be space complete. Therefore, we only transform them into space complete tables whenever there is a need for that.

Chapter 6

The Power of Tabular Expressions

6.1 Introduction

Tabular expressions were successfully adopted because of their convenience especially in making long and complex formulas easily readable. However, their use could be wider, since they very powerful. In this chapter, we want to show the power of tabular expressions. First, we present an application of tabular expressions for three dimensions and higher. To the best of our knowledge, this is the first time such application has been presented [42, 50]. Next, we present a language and a structure for tabular expressions that we came up with. Finally, we explain how tabular expressions can be represented by a lattice and by a vector space respectively.

In the next section, we make a regression to show where tabular expressions get their power in specifying functions, relations, and programs. In Section 6.3, we propose a language and a structure for tabular expressions. Then, in Sections 6.4 and 6.5 we show how tabular expressions can be represented by a lattice and by a vector space respectively. Representing tabular expressions by lattices and vector spaces, allows us to have more potential applications not only in software engineering, but also applications in mathematical

theory. Finally in Section 6.6, we show how tabular expressions can be used to embed programming languages statements.

6.2 Tabular expressions and Turing machines

In this section, we show how programs can be specified by tabular expressions not by looking at programs as a composition of simple, conditional and iterative statements but as at their essence, namely Turing machines. To make the chapter self contained, in the following we give a brief description on Turing machines, however for more details we refer the reader to [46].

A Turing machine is an abstract model of computation that is able to simulate any computer program. It consists of an infinite tape allowing infinite memory capacity. The tape composed of cells has a left end and is unbounded from the right. It has "0" and "1" symbols on the tape, and only one symbol is scanned at a time. It has a read/write head that moves left or right on the tape. A Turing machine is determined by its current state, the scanned symbol in the current cell pointed by the head, and a finite table of instructions called also "transition function" or "action table". These instructions are usually represented by a four-tuple <State_0, Symbol, NextState, Action>. Once the machine is in state State_0, the head is pointing to the current cell containing Symbol, the action has to move to "NextState" with "Action". The possible actions of a Turing machine are either to write a symbol on the tape in the current cell, or to move the head one cell to the left, to the right, or to stay at the same place.

Formally, a Turing machine is specified by a four tuple $(\mathcal{S}, \mathcal{Q}, \mathcal{D}, \delta)$, where $\mathcal{S} = \{s_1, \ldots, s_n\}$ is a finite set of symbols, $\mathcal{Q} = \{q_1, \ldots, q_m\}$ is a finite set of states, $\mathcal{D} = \{d_1, \ldots, d_p\}$ is a

	q_1	q_2	q_3
s_1	(q'_1, s'_1, d)	(q'_2, s'_1, d)	(q'_3, s'_1, d)
s_2	(q'_1, s'_2, d)	(q'_2, s'_2, d)	(q'_3, s'_2, d)

Figure 6.1: A general representation of a tabular expression representing a Turing machine.

finite set of directions, and δ is a transition function defined by:

$$\delta : \mathcal{Q} \times \mathcal{S} \rightarrow \mathcal{Q} \times \mathcal{S} \times \mathcal{D}$$
$$(q, s) \longmapsto (q', s', d)$$

In the following, we show how a Turing machine can be represented by a tabular expression. For instance, a Turing machine with 2 symbols and 3 states can be represented by a tabular expression as the one shown in Figure 6.1, where q'_i designates the new sate, s'_i represents the new symbol, and d depicts the direction.

A multitape Turing machine has a finite number of independent tapes. It is quite similar to a single Turing machine. Although multitape Turing machine seems to be quite powerful, they do not compute more than single tape Turing machines. In fact, any multitape Turing machine can be simulated by a single tape Turing machine. In a multitape Turing machine each tape has a head. At each time the machine reads the scanned symbol pointed by each head, writes the new symbol on each tape, moves each tape head, and make a transition to the next sate.

The transition function of a multitape Turing machine is given by:

$$\delta : \quad \mathcal{Q} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{Q} \times \mathcal{S} \times \mathcal{S} \times \mathcal{D} \times \mathcal{D}$$

$$(q, s_1, s_2) \longmapsto (q', s_1', s_2', d_1, d_2)$$

A Turing machine with multiple tapes can be represented using a tabular expression of higher dimension. For instance, a two-tape Turing machine could be represented by a three dimensional tabular expressions, with three headers. Two headers represent the symbols of each tape, one header represent the state, and a three dimensional grid. In Figure 6.2, we depict the general representation of a tabular expression representing a two-tape Turing machine, where to each state, and each symbol from each tape corresponds a new state, new symbols on each tape, and new moves on each tape head. Tabular expressions are defined as a generalisation of two dimensional tables. However, all the examples found in the literature so far handle only tabular expressions with two dimensions or less. In [30], Jin and Parnas proposed a new type of tabular expression that use redundant information in headers to define several functions, however, the gird is still two-dimensional. In this work, we presented a general representation of a tabular expression representing a two-tape Turing machine. In general, an n tapes Turing machine is represented by $n+1$ dimensional tabular expressions, with n a finite number. To the best of our knowledge, this is the first application of a three dimensions and higher tabular expressions.

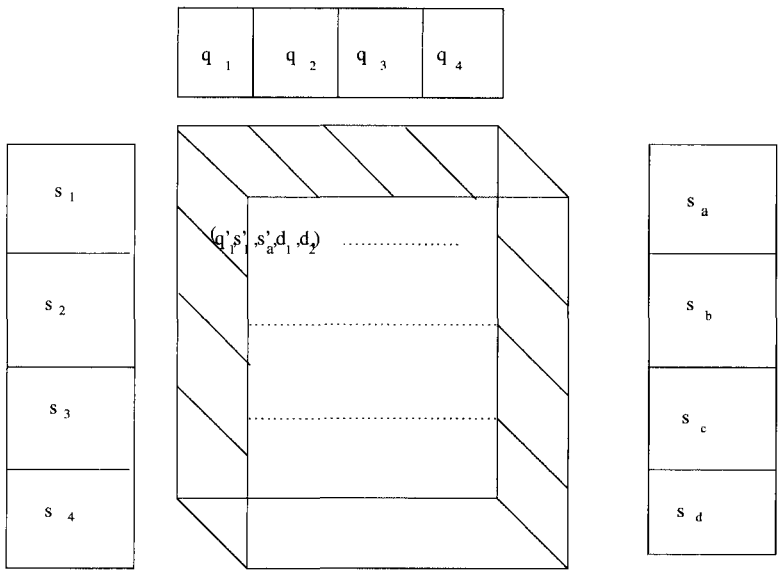


Figure 6.2: A general representation of a tabular expression representing a two-tape Turing machine.

6.3 Language and Structure for Tabular Expressions

6.3.1 Languages and Structures

A *language* $\mathcal{L}$ is defined by:

- A set of symbols.
- A set of function symbols with arity ≥ 1.
- A set of predicate symbols with arity ≥ 1.

Given a language $\mathcal{L}$, a *structure* for $\mathcal{L}$ is specified by:

- A set called carrier set. The elements of the carrier of a structure of a language are called *diagram symbols*.
- A set of functions each with arity n.

– A set of predicates each with arity m.

6.3.2 Diagram Language

In this section we present our *diagram language* which is specified by:

– Four 2-arity predicate symbols called *neighbour*, *flow*, *component*, and *connection* such that:

$$\forall x \forall y, \quad neighbour(x, y) = neighbour(y, x)$$

$$\forall x \forall y, \quad connection(x, y) = connection(y, x)$$

Moreover, the *connected* predicate can be used to generate a *component* predicate as follows:

$\forall x \forall y,$ if *connected* (x, y), then *component* (x, y).

$\forall x \forall y \forall z,$ if *connected* (x, y) and *connected* (y, z), then *component*(x, z).

Component is an equivalence relation, hence it is reflexive, transitive, and symmetric. That is:

$\forall x,$ *component* (x, x) is true

$\forall x \forall y \forall z,$ if *component* (x, y) and *component* (y, z), then *component*(x, z).

$\forall x \forall y,$ if *component* (x, y), then *component* (y, x).

A formula composed of a conjunction of connections is called a *skeleton.*

A formula composed of a conjunction of flows is called a *diagram.*

The skeleton on the equivalence classes of its diagram symbols is called *pattern.*

The diagram on the equivalence classes of its diagram symbols is called *template.*

6.4 The lattice structure

In the following we show how tabular expressions can be interpreted using lattices.
Let Λ be a point lattice in $\mathbb{R}^n$, where $\mathbb{R}$ is the set of real numbers and n is a natural number. The regular lattice Λ is given by:

$$\Lambda = \{\sum_{i=0}^{n} a_i e_i | a_i \in \mathbb{Z}\},$$

where $(e_1, \ldots, e_n)$ is the usual orthonormal basis of $\mathbb{R}^n$ and $\mathbb{Z}$ is the set of integers.

The *neighbour* predicates defined in our diagram language is interpreted as follows. Given two points x and y in the lattice, the *neighbour*(x, y) is true iff x and y are neighbours in the lattice. The *connection* is interpreted as a predicate indicating whether or not two points are neighbour and connected by a line. In Figure 6.3, the expressions $x < 10$ and $x \geq 10$ are neighbours, and connected too. The *flow* is interpreted as a predicate indicating whether or not two components are connected by an arrow.

The *component* predicate allows us to group the connected points into a single class. In the above example, we end up with three classes. In Figure 6.4, the three components are represented by boxes. The three classes represent what we call *pattern.* The *template* is obtained by the *pattern* and the flows which are represented by arrows.

Note that if the dots in the lattice of Figure 6.3 are drawn as cells, we obtain a tabular expression. Thus, a tabular expression can be derived using lattices as shown by the above example. The reason for using points of a lattice instead of cells is their simplicity.

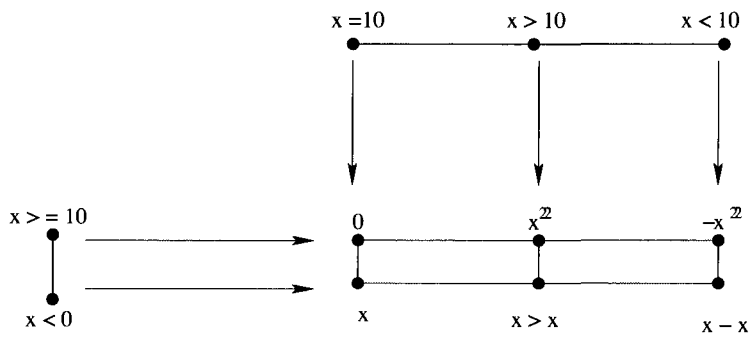


Figure 6.3: An example of a lattice representing the function f.

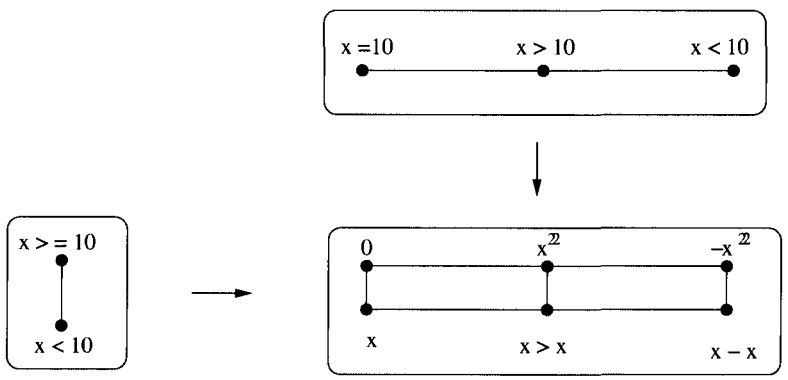


Figure 6.4: An example of a lattice representing the function f, and the components.

6.5 Normal Tabular Expressions as a Vector Space

As we previously mentioned, a normal tabular expression can be interpreted using a point lattice. By making the headers play the role of basis e_i, and the expressions in the grid to play the role of a_i, a point M in the lattice can be written as:

$$M = e_i a_i.$$

Expressions associated to points in the lattice are then represented as:

$$x = \sum_{i=1}^{n} e_i a_i.$$

The meaning of the sum and the product operators depend on the kind of tabular expression we are dealing with.

For example, the point $M_{1,1}$ corresponds to the expression:

$(x \geq 10) \wedge (y = 10) \odot \{0\}$.

The whole expression of the lattice corresponds to:

$$\begin{aligned}
& (x \geq 10) \wedge (y = 10) \odot \{0\} \\
\oplus\ & (x \geq 10) \wedge (y > 10) \odot \{y^2\} \\
\oplus\ & (x \geq 10) \wedge (y < 10) \odot \{-y^2\} \\
\oplus\ & (x < 10) \wedge (y = 10) \odot \{x\} \\
\oplus\ & (x < 10) \wedge (y > 10) \odot \{x + y\} \\
\oplus\ & (x < 10) \wedge (y < 10) \odot \{x - y\}
\end{aligned}$$

The sum and the product operators are represented by the symbols $\oplus$ and $\odot$ respectively.

$C_1 \wedge \neg(C_2 \vee C_3 \vee \cdots \vee C_n)$	$C_2 \wedge \neg(C_1 \vee C_3 \vee \cdots \vee C_n)$	...	$\neg(C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_n)$
p_1	p_2	...	p_{n+1}

Figure 6.5: Tabular representation of if statements.

6.6 Tabular expressions and programming languages

The framework introduced above allows tabular expressions to embed any language expressions including programming language statements.

Therefore we are able to represent conditional statements of the form if - elif -... else. Also in [35], *if* statements were represented by tabular expression. For example a statement of form

$$\text{if } c_1 \text{ then } p_1 \text{ elif } c_2 \text{ then } p_2, \ldots \text{ else } p_{n+1}$$

is represented by the one dimensional table drawn in Figure 6.5.

In this chapter, we propose to express *for* loops in a tabular expression form. For that purpose, we need to define a *tabular counter* TC. A tabular counter holds the next expression to evaluate. It can refer to another table. Also since it introduces recursion, it allows us to represent nested tables with ease. For example in the normal table H shown in Figure 6.6, we see that in cell $(2, 2)$, the tabular counter holds the expression $H(x, y)$ that should be evaluated, and which introduces a form of recursion.

A *for* loop can be generalized using the *if... elsif... else* and *goto* constructs. The *goto* points to the beginning of *if... elsif... else*.

For example the following loop:

```
for (int i1=0, ..., in=0; C1(i1, ..., in); ++i1,...,++in) {
```

	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x + y$ $TC = H(x, y)$	$x - y$

Figure 6.6: An example of a normal table H.

```
        stts;
}
```

can be written as:

```
int i1=0, ..., in=0;
L1: if (C1(i1, ..., in)) {
        stts;
        ++i1; ...;++in;
        goto L1;
    }
```

The general construct can be translated as follows:

```
int i1=0, ..., in=0;
L1:   if (C1(i1, ..., in)) {
        stts1;
        goto L1;
```

```
        }
 L2:    elsif (C2(i1, ..., in)) {
            stts2;
            goto L2;
        }
        .
        .
        .
        }
Lm:     elsif (Cm(i1, ..., in)) {
            sttsm;
            goto Lm;
        } else {
           stts0;
        }
```

As a particular example, the *for* loop presented above can be expressed as the following tabular expression given in Figure 6.7.

$C_1 \wedge \neg(C_2 \vee C_3 \vee \cdots \vee C_n)$	$C_2 \wedge \neg(C_1 \vee C_3 \vee \cdots \vee C_n)$	...	$\neg(C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_n)$
$stts_1$ $TC = T(i_1, i_2, \ldots, i_n)$	$stts_2$ $TC = T(i_1, i_2, \ldots, i_n)$	... ...	$stts_{n+1}$ $TC = T(i_1, i_2, \ldots, i_n)$

Figure 6.7: Tabular representation of for loops.

Chapter 7

Conclusion

In this thesis, we converted SCR tables into tabular expressions because tabular expressions have a precise and rich formal semantics, while SCR is limited with regards to its semantics which is not well defined [19]. The conversion of SCR tables into tabular expressions that we proposed, allows the SCR converted tables to inherit the semantics of tabular expressions. The algorithms that we propose are quite efficient. Many advantages are gained with the conversion that we presented. The tables are more readable, and can be easily interpreted even by people who do not have the knowledge of the domain. Furthermore, we removed previous ambiguous symbols (e.g. primed notations and prefixed notations with "@" symbols), and we proposed a new way to model SCR events with first order logic and with propositional logic. Hence, by improving SCR semantics, there are many tasks that can now be performed such as enabling the method to support table composition and decomposition, and improving the toolset supporting the SCR method.

For tabular expressions, we improved both syntax and semantics. Now that, tabular expressions are seen as a stack of atomic expressions and operators that are applied to them, one can easily have a build up of tables from atomic ones. This buildup view enhances building tools supporting the semantics. Also, we introduced a partial order and a

refinement ordering relation defined on tabular expressions, and we produced an algebra for tabular expressions. In our algebraic model, we introduced new operators that allow us to compose and decompose tabular expressions. Our algebra might be useful to solve the open research problem which consists in composing a relation R from its parts R_α [26], and thus coming up with a universal composition. A novelty in our work is that the composition of two tables is not always a single table, but could be a set of tables. We introduced a partial order relation, and a refinement ordering relation defined on tabular expressions.

Moreover, we presented an application of a tabular expressions for three dimensions and higher. We showed that tabular expressions are very suitable for representing Turing machines. In fact, the tabular representation of Turing machines increases their readability. Also, to the best of our knowledge this is the first time that an application for tabular expressions with dimensions greater than or equal to three has been proposed. We also proposed a language and a structure for tabular expressions. Finally, we developed a lattice and vector space representation for tabular expressions.

Future work

There is much subsequent work that can be carried on from ours. In the following, we give some ideas for future work. In [17], Heitmeyer showed how the SCR model could be extended to specify hybrid systems (systems containing both discrete and continuous variables). However, time was not added to the general SCR model, and concurrency was not discussed. Therefore, as a future work, we propose adding time to the general SCR model so it can handle concurrent systems. For SCR events, we suggest that the conditioned events will be executed by guarded commands where the condition and action will be representing events.

To handle the algebra of tabular expressions that we introduced in a more abstract way,

we propose extending our work by adopting category theory. Category theory offers the ease of manipulating mathematical structures and the relationship between them. In our category, the objects will be the classes of tabular expressions, and the morphisms will be the inner and Kronecker operators. Although a lot of work has been done and many modules have been implemented, it will be very helpful to have a quality professional tool supporting tabular expressions [51]. We suggest that the new model that we have proposed for tabular expressions, we suggest that it will be applied not only at the requirements level, but at every phase of the software development process model.

Appendix A

In this appendix, we give examples to show how to extend Janicki's work to compose tables instead of composing cells of one table. We give examples of composition of normal tables, inverted tables, vector tables, and decision tables.

The normal table presented in Figure A.3 is the union of the normal tables f_1 and f_2 drawn in Figures A.1, and A.2 respectively.

The union of the inverted tables g_1 and g_2 drawn in Figures A.4, and A.5 respectively, gives the inverted table g presented in Figure A.6.

The union of the vector tables G_1 and G_2 drawn in Figures A.7, and A.8 respectively, gives the vector table G presented in Figure A.9.

The union of the generalized decision tables h_1 and h_2 drawn in Figures A.10, and A.11 respectively, gives the generalized decision table G presented in Figure A.12.

	$y = 10$
$x \geq 0$	0
$x < 0$	x

Figure A.1: The Normal Table f_1

	$y = 10$	$y > 10$
$x \geq 0$	0	y^2
$x < 0$	x	$x + y$

Figure A.2: The Normal Table f_2

	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x + y$	$x - y$

Figure A.3: The Normal Table $f = f_1 \cup f_2$

	$x+y$	$x*y$
$y \geq 0$	$x < 0$	$x \geq 0$
$y < 0$	$x < y$	$y \leq x$

Figure A.4: The inverted table g_1

	$x-y$	$y-x$
$y \geq 0$	$0 \leq x < y$	$x \geq 0$
$y < 0$	$y \leq x < 0$	$x \geq y$

Figure A.5: The inverted table g_2

	$x+y$	$x*y$	$x-y$	$y-x$
$y \geq 0$	$x < 0$	$x \geq 0$	$0 \leq x < y$	$x \geq y$
$y < 0$	$x < y$	$y \leq x$	$y \leq x < 0$	$x \geq 0$

Figure A.6: The inverted table $g = g_1 \cup g_2$

	$x_2 \leq 0$	$0 < x_2 < y$
$y_1 =$	$x_1 - x_2$	$-x_1 + x_2$
$y_2 \mid$	$y_2 x_1 + x_2 = y_2^2$	$-x_1 + x_2 y_2 = y_2^2$
$y_3 \mid$	$y_3 + x_1 x_2 = y_3^3$	$y_3 + x_1 x_2 = y_3^3$

Figure A.7: The vector table G_1

	$x_2 \geq y$
$y_1 =$	$x_1 + x_2$
$y_2 \mid$	$x_1 + y_2 = y_2^2$
$y_3 \mid$	$y_3 = x_1$

Figure A.8: The vector table G_2

	$x_2 \leq 0$	$0 < x_2 < y$	$x_2 \geq y$
$y_1 =$	$-x_1 - x_2$	$-x_1 + x_2$	$x_1 + x_2$
$y_2 \mid$	$y_2 x_1 - x_2 = y_2^2$	$-x_1 + x_2 y_2 = y_2^2$	$x_1 + y_2 = y_2^2$
$y_3 \mid$	$y_3 + x_1 x_2 = y_3^3$	$y_3 + x_1 = y_3^2$	$y_3 = x_1$

Figure A.9: The vector table $G = G_1 \cup G_2$

	$x_1 + x_2$	$x_1 - x_2$
x_1x_2	$\# < 20$	$\# \geq 20$
x_1/x_2	$\# > 30$	$\# < 30$

Figure A.10: The generalized decision table h_1

	x_1x_2
x_1x_2	$\# \neq 30$
x_1/x_2	$\# = 30$

Figure A.11: The inverted table h_2

	$x_1 + x_2$	$x_1 - x_2$	x_1x_2
x_1x_2	$\# < 20$	$\# \geq 20$	$\# > 30$
x_1/x_2	$\# > 30$	$\# < 30$	$\# = 30$

Figure A.12: The generalized decision table $h = h_1 \cup h_2$

Appendix B

Here, we show how function tables fit in the general framework of tabular expressions. There are four kinds of function tables: vertical condition tables, horizontal condition tables, structured decision tables, and state transition tables. We take examples of function tables borrowed from [36], and we show how they correspond to tabular expressions.

The vertical condition table can be seen as a one dimensional normal table with one header and one grid as shown in Figure B.1. It correspond to Type 1 of Janicki's classification [25, 26, 28]. For instance the vertical condition table depicted in Figure B.2 is presented by the tabular expression drawn in Figure B.3.

The horizontal condition table can be seen as two dimensional inverted table. It corresponds to Type 2a of Janicki's classification as shown in Figure B.4. For instance, the horizontal condition table depicted in Figure B.8 is presented by the tabular expression

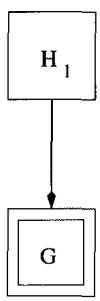


Figure B.1: Type1. One header and one grid.

VCT: trip

	(m-level >level-limit) AND (m-enable = e-enabled)	NOT(m-level > k-level-limit) OR NOT (m-enable = e-enabled)
f-trip	e-tripped	e-not-tripped

Figure B.2: Vertical condition table

(m-level >level-limit) AND (m-enable = e-enabled)	NOT(m-level > k-level-limit) OR NOT (m-enable = e-enabled)

(f-trip=e-tripped)	NOT (f-trip=e-not-tripped)

Figure B.3: Tabular expression corresponding to VCT table.

drawn in Figure B.6.

The structured decision tables may have conditions macros that come with the table helping shortening the cells content. In this example, the condition macros are:

w-trip-rng[m-ai, f-sp]

hitrp : m-ai $\geq$= f-sp

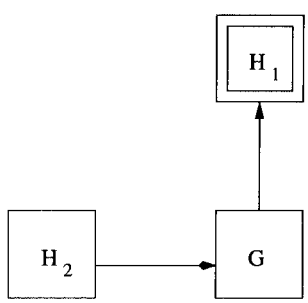


Figure B.4: Type2a. Two headers and one grid.

HCT: foo-fee

	Result	
Conditions	f-foo	f-fee
m-Trip[1] = 1 AND m-Trip[1] $\neq$1	e-tripped	e-not-tripped
m-Trip[1] $\neq$1 AND m-Trip[2] = 1	e-tripped	e-not-tripped
m-Trip[1] $\neq$1 AND m-Trip[2] $\neq$ 1	e-not-tripped	e-not-tripped
m-Trip[1] =1 AND m-Trip[2] = 1	e-tripped	e-tripped

Figure B.5: Horizontal condition table

ddbnd : (m-ai < f-sp) AND (m-ai $\geq$ (f-sp - k-db))

notrp : m-ai < (f-sp - k-db)

The structured decision table corresponds to decision tables, which is type 2a in Janicki's classification as shown in Figure B.4. In this example, the structured decision table depicted in Figure B.8 can be seen as the tabular expression shown in Figure B.9.

The state transition table (STT) can be seen as a two dimensional normal table with two headers and one grid as shown in Figure B.10. It corresponds to Type 1 of Janicki's classification. For instance, the state transition table depicted in Figure B.11 is presented by the tabular expression drawn in Figure B.12.

In [1], Abraham proposed that VCT and HCT correspond to vector tables. However, we think that VCT, and HCT fit better into normal table, and inverted table respectively as we proposed. Indeed, vector tables are useful to describe a function whose range is a set of tuples [1]. However, this is not the case for HCT, and VCT (counterexamples are given above).

	f-foo	f-fee
m-Trip[1] = 1 AND m-Trip[1] $\neq$ 1	e-tripped	e-not-tripped
m-Trip[1] $\neq$ 1 AND m-Trip[2] = 1	e-tripped	e-not-tripped
m-Trip[1] $\neq$ 1 AND m-Trip[2] $\neq$ 1	e-not-tripped	e-not-tripped
m-Trip[1] = 1 AND m-Trip[2] = 1	e-tripped	e-tripped

Figure B.6: Tabular expression corresponding to HCT table.

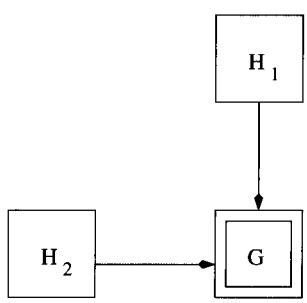


Figure B.7: Type3. Two headers and one grid.

SDT: trip

Condition Statements	1	2	3	4
w-trip-mg[m-ai, f-sp1]	hitrp	ddbnd	ddbnd	notrp
$f\text{-}trip_{-1}$ = e-tripped	-	T	F	-
Action Statements				
f-trip = e-tripped	X	X		
f-trip = e-not-tripped			X	X

Figure B.8: Structured decision table

	f-trip=e-tripped	f-trip=e-tripped	f-trip=e-not-tripped	f-trip=e-not-tripped
w-trip-mg[m-ai, f-sp1]	hitrp	ddbnd	ddbnd	notrp
$f\text{-}trip_1$=e-tripped		True	False	

Figure B.9: Tabular expression corresponding to a decision table

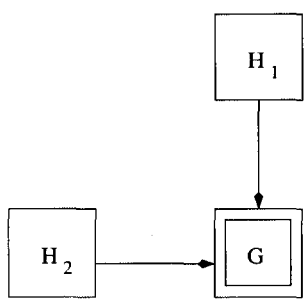


Figure B.10: Type3. Two headers and one grid.

STT: f-digitalwatch

Transition Condition / **Previous State** ↓	m-select = e-pressed	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-pressed)	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-unpressed)	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-unpressed)	(m-select = e-unpressed) AND (m-start-stop = e-pressed) AND (m-reset e-unpressed)
e-time	e-in-time	e-time	e-time	e-time	e-time
e-in-time	e-in-time	e-in-time	e-in-time	e-in-time	e-zero
e-zero	e-stopwatch	e-zero	e-running1	e-running2	e-running2
e-running1	e-stopwatch	e-running1	e-running1	e-running2	e-running2
e-running2	e-stopwatch	e-stopped1	e-stopped1	e-running2	e-running2
e-stopped1	e-stopwatch	e-stopped1	e-stopped1	e-stopped	e-stopped2
e-stopped2	e-stopwatch	e-stopped2	e-running1	e-stopped	e-stopped2
e-stopped	e-stopwatch	e-stopped	e-stopped	e-stopped	e-zero
e-stopwatch	e-stopwatch	e-time	e-time	e-time	e-time

Figure B.11: State transition table

	(m-select =e-pressed)	(m-select =e-unpressed) AND (m-start-stop =e-unpressed)	(m-select =e-unpressed) AND (m-start-stop =e-unpressed)	(m-select =e-unpressed) AND (m-start-stop =e-unpressed)	(m-select =e-unpressed) AND (m-start-stop =e-unpressed)
e-time	e-in-time	e-time	e-time	e-time	e-time
e-in-time	e-in-time	e-in-time	e-in-time	e-in-time	e-zero
e-zero	e-stopwatch	e-zero	e-running-1	e-running-2	e-running-2
e-running-1	e-stopwatch	e-running-1	e-running-1	e-running-2	e-running-2
e-running-2	e-stopwatch	e-stopped1	e-stopped1	e-running-2	e-running-2
e-stopped1	e-stopwatch	e-stopped1	e-stopped1	e-stopped	e-stopped2
e-stopped2	e-stopwatch	e-stopped2	e-running-1	e-stopped	e-stopped2
e-stopped	e-stopwatch	e-stopped	e-stopped	e-stopped	e-zero
e-stopwatch	e-stopwatch	e-time	e-time	e-time	e-time

Figure B.12: Tabular expression corresponding to STT table

Bibliography

[1] R. Abraham. Evaluating generalized tabular expressions in software documentation. Master's thesis, Dept. of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, Canada, 1997.

[2] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software requirements for the A-7E aircraft. Technical report, NRL-919. Naval Research Lab., Washigton, DC, 1992.

[3] Aonix. Software through pictures. www.aonix.com/stp.html, 2006. Last accessed October 29, 2009.

[4] B. Bohem. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):pages 75–88, January 1984.

[5] I. Bourguiba and R. Janicki. Table-based specification techniques. In *International Conference on Computers & Industrial Engineering 2009 (CIE39)*, pages 1520–1525, Troyes, France, July 6-9, 2009. IEEE.

[6] I. Bourguiba and R. Janicki. Tabular Expressions vs Software Cost Reduction. In H. Arabnia and H. Reza, editors, *Proceedings of the 2009 International Conference on Software Engineering Research & Practice (SERP 2009)*, volume 2, pages 403–407, Las Vegas, USA, July 13-16, 2009. 2009 CSREA Press.

[7] J. Desharnais, M. Frappier, R. Khedri, and A. Mili. Integration of sequential scenarios. *IEEE Transactions on Software Engineering*, 24(9):pages 695–708, September 1998.

[8] J. Desharnais, R. Khedri, and A. Mili. Interpretation of tabular expressions using arrays of relations. *In E. Orlowska and A. Szalas editors. Relational Methods for Computer Science Applications*, Vol. 65 of Studies in Fuzziness and Soft Computing:pages 3–14, 2001. Springer-Physica Verlag.

[9] J. Desharnais, R. Khedri, and A. Mili. Representation, validation and integration of scenarios using tabular expressions. *Formal Methods in System Design*, 2005. To appear.

[10] G. A. D.L. Parnas and J. Madey. Assesment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):pages 19–198, 1991.

[11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):pages 231–274, June 1987.

[12] M. Heimdahl and B. J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *High-Assurance Systems Engineering, (HASE '96)*, pages 252–256, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[13] M. Heimdahl and N. G. Leveson. Completeness and consistency analysis of state-based requirements. In *International Conference on Software Engineering. Proceedings of the 17th international conference on Software engineering*, pages 3–14, 1995.

[14] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and

analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, June 1995.

[15] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):pages 231–261, 1996.

[16] C. Heitmeyer, J. Kirby, and B. Labaw. Applying the SCR requirements method to a weapons control panel: An experience report. In *Proceedings of FMSP'98. Second Workshop on Formal Methods in Software Practice*, Clearwater Beach, FL, USA, Narch 4-5, 1998. ACM.

[17] C. L. Heitmeyer. Requirements specifications for hybrid systems. *Hybrid Systems Workshop III, Lecture Notes in Computer Science*, 1066:pages 304–314, 1996. Springer-Verlag.

[18] C. L. Heitmeyer. Formal methods for specifying, validating, and verifying requirements. *Journal of Universal Computer Science*, 13(5):pages 606–618, May28, 2007.

[19] C. L. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *Computer Systems Science and Engineering*, 20(1):pages 19–35, January 2005.

[20] C. L. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of 12th Annual Conference on Computer Assurance (COMPASS '97)*, pages 35–47. IEEE Computer Society, June 1997.

[21] K. Henninger, J. Kallander, D. Parnas, and J. Shore. Software requirements for the

A-7E Aircraft. NRL Memorandum. Technical report, 3876. U.S. Naval Research Lab, 1978.

[22] D. N. Hoover and Z. Chen. Tablewise, a decision table tool. In *Proceedings of the 9th Annual Conference on Computer Assurance COMPASS'95. Systems Integrity, Software Safety and Process Security*, pages 97–108, Gaithersburg, MD, USA, June 25-29, 1995.

[23] R. B. Hurlay. *Decision Tables in Software Engineering*. Prentice Hall PTR, Van Nostrand, New York, 1983.

[24] R. Janicki. Remarks on mereology of direct products and relations. In *J. Desharnais, M. Frappier, W. MacCaull (eds.), Relational Methods in Computer Science*, pages 65–84, 2002.

[25] R. Janicki. Towards a formal semantics of Parnas tables. In *17th International Conference on Software Engineering*, pages 231–240, April 23-30, 1995.

[26] R. Janicki and R. Khedri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 39(2):pages 189–213, March 2001.

[27] R. Janicki, D. Parnas, and J. Zucker. Tabular representations in relational documents. In *Proceedings of Relational Methods in Computer Science*, pages 184–196. Springer Verlag, 1997.

[28] R. Janicki and A. Wassyng. Tabular expressions and their relational semantics. *Fundamenta Informaticae*, 67(4):pages 343–370, 2005.

[29] R. Janicki and A. Wassyng. On tabular expressions. In *D.A. Stewart, ed. Proceedings of CASCON 2003*, pages 38–52, Ontario, Canada, October 2003.

[30] Y. Jin and D. Parnas. Defining the meaning of tabular mathematical expressions. *Science of Computer Programming*, 75(11):pages 980–1000, November 2010.

[31] W. Kahl. Compositional syntax and semantics of tables. Technical report, Software Quality Research Laboratory Report no. 15. McMaster Univeristy, October 12, 2003.

[32] R. Khedri. Requirements scenarios formalization technique: N versions towards one good version. *Electronic Notes in Theoretical Computer Science*, 44(3):pages 112–135, 2003.

[33] R. Khedri and I. Bourguiba. Formal derivation of functional architectural design. In I. C. Society, editor, *Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 356–365, Beijing, China, September 28-30, 2004.

[34] R. Khedri and I. Bourguiba. Requirements scenarios based system-testing. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 252–257, Alberta, Canada, June 20-24, 2004.

[35] M. Mohrenschildt. Algebraic composition of function tables. *Formal Aspects of Computing*, 12:pages 41–51, 2000.

[36] G. Moum. Procedure for the specification of software requirements for safety critical software. Technical report, CANDU Computer sytems Engineering Centre of Excellence Procedure, Report CE-1001-PROC Rev.2, April 2009.

[37] G. O'regan. *Mathematical Approaches to Software Quality*. Springer, 2006.

[38] D. Parnas. Tabular representation of relations. Technical Report CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, 1992.

[39] D. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):pages 856–862, September 1993.

[40] D. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):pages 41–61, 1995.

[41] D. Parnas and D. Peters. An easily extensible toolset for tabular mathematical expressions. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems. Lecture Notes in Computer Science*, volume 1579, pages 345–359, 1999.

[42] D. Peters. Private communication to R. Janicki, 2007.

[43] T. Pressburger. Software engineering research/developer collaborations in 2005. Technical report, NASA Technical Report, March 7, 2006.

[44] T. Rothamel, Y. A. Liu, C. L. Heitmeyer, and E. I. Leonard. Generating optimized code from SCR specifications. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference for Languages, Compilers, and Tools for Embedded Systems, LCTES 2006*, pages 135–144, Ottawa, Ontario, Canada, June 2006.

[45] E. Sekerinski. Exploring tabular verification and refinement. *Formal Aspects of Computing*, 15(2-3):pages 215–236, November 2003. Springer-Verlag.

[46] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[47] J. Thompson, M. Whalen, and M. Heimdahl. Requirements capture and evaluation in Nimbus: The light-control case study. *Journal of Universal Computer Science*, 6(7):pages 731–757, July 2000.

[48] A. J. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical report, 90-276. Queen's University, Telecommunications Research Institute of Ontario (TRIO). Reprinted as CRL Report 242 (Communication Research Laboratory, McMaster University, Hamilton, Ontario, Canada, February 1992.

[49] F. Wanger. VFSM executable specification. In *International Conference on Computer Systems and Software Engineering*, The Netherlands, May 4-8, 1992. IEEE Computer Society.

[50] A. Wassyng. Private communication, December 2010.

[51] A. Wassyng and R. Janicki. Tabular expressions in software engineering. In *Proceedings of International Conference on Software and Systems Engineering and their Applications*, volume 4, pages 1–16, Paris, December 2003.

[52] A. Wassyng and M. Lawford. Software tools for safety-critical software development. *International Journal on Software Tools for Technology. Special section on the industrialization of formal methods: a view from formal methods*, 8(4/5):pages 337–354, 2006. Springer-Verlag.

[53] A. Wassyng and M. Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. *Proceedings FME 2003: Formal Methods, International Symposium of Formal Methods. Lecture Notes in Computer Science*, 2805:pages 133–153, September 8-14, 2003.

[54] Y. Yang and R. Janicki. Modelling concurrency with tabular expressions. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04)*, volume 2, pages 455–461, June 2004.