Parnas Tables: A Practical Formalism

Joanne M. Atlee
Department of Computer Science
University of Waterloo
Software is increasingly used to control or manage **critical systems**

**safety-critical systems** in which a failure can lead to loss of life
(e.g., medical devices, nuclear power plants, airplanes, cars, trains)

**mission-critical systems** in which a failure can cause significant loss of property
(e.g., spacecraft, satellites, manufacturing, security systems, financial systems)
How to Achieve Confidence in Critical Software?

There are several complementary verification activities.

1. **Review software documents** (requirements, design, code)
   - to **reveal errors early** in the development process, when they are easier to correct (cf. testing, code reviews)
   - to **exhaustively examine** an artifact (cf. testing)
   - to **locate defects** (cf. testing)

2. **Test code systematically** to confirm expected behaviour to **evaluate the final product** in its operational environment (cf. reviews)

3. **Test code randomly** to reveal unexpected behaviour to help assess the **software’s reliability** (cf. reviews)

4. **Perform hazard analysis** to detect and avoid causes of failures

This talk focuses on writing and reviewing software documentation
Software Documentation

**Software Documentation** - technical documents that explain a software system’s

- **requirements** - required goals of system
- **specification** - specified functionality of system
- **design** - decomposition of system into modules, and
  - specified functionality of each module
- **descriptions** - actual functionality of program fragments
(Im)Precise Documentation

If one does not have a precise definition of a system’s desired behaviour, how can one possibly expect to evaluate that the implemented system meets its requirements?
Mathematical Documentation

In other engineering disciplines, “precise documentation” means **mathematical definitions**

- **unambiguous**

- **consistency, completeness**, and other desired properties are well-defined and can be checked

- **composition** of components is well-defined
Mathematical Documentation

In contrast, mathematical methods are not widely used to document software because software can implement a function

- that has many discontinuities
- whose domain and range are tuples of distinct types

making it difficult to express behaviour in a compact mathematical definition.

Example: Elevator Direction

![Graph showing elevator direction over time]

- Time
- Direction: up, down
Elevator Example

An elevator’s direction depends on its **current direction** (dir), the **floor** that it is on (loc), and what **requests** are pending (Req[[]]).

It travels in a given direction until

- there are **no** more pending requests in the **current direction**
- and there are pending requests in the **opposite direction**.

\[
\text{dir} : \{\text{Up, Down}\}
\]
\[
\text{loc}: \{1..\text{n}\}
\]
\[
\text{Req}[1..\text{n}] : \text{boolean}
\]

\[
\text{ElevDir}(\text{dir},\text{loc}, \text{Req}[[]]) = \begin{cases} 
\text{Up} & (\text{dir} = \text{Up} \land \exists f.(f \geq \text{loc} \land \text{Req}[f])) \lor \\
& (\text{dir} = \text{Down} \land \neg \exists f.(\leq \text{loc} \land \text{Req}[f]) \land \\
& \exists f.(f > \text{loc} \land \text{Req}[f])) \\
\text{Down} & (\text{dir} = \text{Down} \land \exists f.(\leq \text{loc} \land \text{Req}[f])) \lor \\
& (\text{dir} = \text{Up} \land \neg \exists f.(\geq \text{loc} \land \text{Req}[f]) \land \\
& \exists f.(f < \text{loc} \land \text{Req}[f])) \\
\text{dir} & \text{otherwise}
\end{cases}
\]
Practical Formalisms are notations with precise semantics that can be read and reviewed by domain experts and software professionals.

- They have a formal, mathematical model
- They encourage the use of separation of concerns and abstraction to decompose and simplify a problem
- They have diagrammatic constructs for expressing functions and relations
- …that encourage the writer to consider completeness

Examples: Statecharts, SDL, Petri-Nets, Parnas Tables, SCR, CoRE, RSML, Tablewise
Parnas Tables

Parnas Tables use **tabular** constructs to organize mathematical expressions, where

- rows and columns separate an expression into **cases**
- each table entry specifies either the **result value for some case** or a **condition that partially identifies some case**

**Example: Inverted Table**

```
<table>
<thead>
<tr>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pred1,A</td>
<td>Pred2,A</td>
<td>Pred3,A</td>
</tr>
<tr>
<td>Pred1,B</td>
<td>Pred2,B</td>
<td>Pred3,B</td>
</tr>
</tbody>
</table>
```

\[
F_{2,A} \equiv \text{if } \text{Pred}_A \land \text{Pred}_{2,A} \text{ then Result } = \text{Value}_2
\]

\[
F \equiv \bigcup_{j=A,B} F_{i,j}
\]
ElevDir(dir,loc,Req[]) =

<table>
<thead>
<tr>
<th>dir</th>
<th>Up</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>$\exists f.(f\geq loc \land Req[f])$</td>
<td>$\neg \exists f.(f\geq loc \land Req[f]) \land \exists f.(f&lt;loc \land Req[f])$</td>
</tr>
<tr>
<td>Down</td>
<td>$\neg \exists f.(f\leq loc \land Req[f]) \land \exists f.(f&gt;loc \land Req[f])$</td>
<td>$\exists f.(f\leq loc \land Req[f])$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dir</th>
<th>Up</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>$(dir = Up \land \exists f.(f\geq loc \land Req[f])) \lor$</td>
<td>$(dir = Down \land \neg \exists f.(f\leq loc \land Req[f]) \land \exists f.(f&gt;loc \land Req[f]))$</td>
</tr>
<tr>
<td>Down</td>
<td>$(dir = Down \land \exists f.(f\leq loc \land Req[f])) \lor$</td>
<td>$(dir = Up \land \neg \exists f.(f\geq loc \land Req[f]) \land \exists f.(f&lt;loc \land Req[f]) \land \exists f.(f&lt;loc \land Req[f])$</td>
</tr>
<tr>
<td>dir</td>
<td>otherwise</td>
<td></td>
</tr>
</tbody>
</table>
Multiple Table Types

The term **Parnas Tables** actually refers to a collection of **table types** and **abbreviation strategies** for organizing and simplifying functional and relational expressions.

An expression can usually be represented in several table types. The documenter’s goal is to choose (or create) a table format that produces a **simple, compact representation** for that expression.

**Example: Normal Table**

\[
\begin{array}{ccc}
  & \text{Pred}_1 & \text{Pred}_2 \\
\text{Pred}_A & \text{Value}_{1,A} & \text{Value}_{2,A} \\
\text{Pred}_B & \text{Value}_{1,B} & \text{Value}_{2,B} \\
\end{array}
\]

\[
F_{2,A} \equiv \text{if } \text{Pred}_A \land \text{Pred}_2 \\
\text{then Result} = \text{Value}_{2,A}
\]

\[
F \equiv \bigcup_{j=A..B}^{i=1..3} F_{i,j}
\]
ElevDir(dir, loc, Req[]) =

<table>
<thead>
<tr>
<th>( \exists f. (f \geq \text{loc} \land \text{Req}[f]) )</th>
<th>true</th>
<th>( \exists f. (f \leq \text{loc} \land \text{Req}[f]) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

ElevDir(dir, loc, Req[]) =

\[
\begin{align*}
\text{dir} & \quad \text{Up} & \\
\text{Up} & \left( \text{dir} = \text{Up} \land \exists f. (f \geq \text{loc} \land \text{Req}[f]) \right) \lor \\
& \left( \text{dir} = \text{Down} \land \neg \exists f. (f \leq \text{loc} \land \text{Req}[f]) \land \right. \\
& \quad \quad \exists f. (f > \text{loc} \land \text{Req}[f]) \\
\text{Down} & \left( \text{dir} = \text{Down} \land \exists f. (f \leq \text{loc} \land \text{Req}[f]) \right) \lor \\
& \left( \text{dir} = \text{Up} \land \neg \exists f. (f \geq \text{loc} \land \text{Req}[f]) \land \\
& \quad \quad \exists f. (f < \text{loc} \land \text{Req}[f]) \\
dir & \quad \text{otherwise}
\end{align*}
\]
## Decision Table

A **Decision Table** is useful for representing a function or relation whose **domain is a tuple** (possibly of distinct types). One dimension of the table itemizes the elements of the domain tuple.

### Expression Diagram

<table>
<thead>
<tr>
<th>Expr_A</th>
<th>Expr_1,A</th>
<th>Expr_2,A</th>
<th>Expr_3,A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr_A</td>
<td>Expr_1,B</td>
<td>Expr_2,B</td>
<td>Expr_3,B</td>
</tr>
</tbody>
</table>

\[
F_2 \equiv \text{if Expr}_A = \text{Expr}_2,A \\
\text{and Expr}_B = \text{Expr}_2,B \\
\text{then Result} = \text{Value}_2
\]

\[
F = \bigcup_{i=1..3} F_i
\]

---

**ElevDir(dir,loc,Req[]) =**

<table>
<thead>
<tr>
<th>dir</th>
<th>Up</th>
<th>Down</th>
<th>Down</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists f. (f \geq \text{loc} \land \text{Req}[f]) )</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>( \exists f. (f \leq \text{loc} \land \text{Req}[f]) )</td>
<td>true</td>
<td>false</td>
<td>---</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Up</th>
<th>Up</th>
<th>Down</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>false</td>
<td>---</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
## Vector Tables

A **Vector Table** is useful for representing a function or relation whose **range is a tuple** (possibly of distinct types). One dimension of the table itemizes the elements of the range tuple.

\[
\begin{array}{|c|c|c|}
\hline
\text{Pred}_1 & \text{Pred}_2 & \text{Pred}_3 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Var}_A' & \text{Value}_{1,A} & \text{Value}_{2,A} & \text{Value}_{3,A} \\
\hline
\text{Var}_B' & \text{Value}_{1,B} & \text{Value}_{2,B} & \text{Value}_{3,B} \\
\hline
\end{array}
\]

\[F_{2,A} \equiv \text{if Pred}_2 \quad \text{then Var}_A' = \text{Value}_{2,A}\]

\[F \equiv \bigotimes_{i=A}^{B} \bigcup_{j=1}^{3} F_{i,j}\]

---

<table>
<thead>
<tr>
<th>Req[loc]</th>
<th>(\neg\text{Req[loc]})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg \exists f.\text{Req}[f])</td>
<td>(\exists f.(f&gt;\text{loc} \land \text{Req}[f]) \land \neg \exists f.(f&lt;\text{loc} \land \text{Req}[f]))</td>
</tr>
<tr>
<td>(\exists f.(f&lt;\text{loc} \land \text{Req}[f]) \land \neg \exists f.(f&gt;\text{loc} \land \text{Req}[f]))</td>
<td>(\exists f.(f&lt;\text{loc} \land \text{Req}[f]) \land \exists f.(f&gt;\text{loc} \land \text{Req}[f]))</td>
</tr>
</tbody>
</table>

| dir'| | dir | dir | Up | Down | dir |
| speed'| | idle | idle | moving | moving | moving |
Properties of Parnas Tables

For each table type, there are rules for identifying

• **distinct cases (subfunctions, subrelations)**
• **mission cases (incompleteness)**
• **conflicting cases (inconsistency)**

<table>
<thead>
<tr>
<th>dir</th>
<th>Up</th>
<th>Down</th>
<th>??</th>
<th>Down</th>
<th>Up</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>∃ f. (f ≥ loc ∧ Req[f])</td>
<td>Up</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>---</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>∃ f. (f ≤ loc ∧ Req[f])</td>
<td>---</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
A-7E Experience

A-7E U.S. Naval Aircraft:
Onboard flight software for an operational naval aircraft
(navigation, navigational update, weapons delivery)

Project:
An experiment, funded by the Naval Research Laboratory (NRL),
to evaluate state-of-the-art software engineering methods

Experience:
• Introduced the first Parnas Tables (without formal definition)
in the Software Requirements Specification (SRS)

• SRS was reviewed by domain experts, pilots, who found hundreds detail errors
A-7E Experience

Since Then:

• The software manager for the **A-7D Air Force aircraft** had his team modify the A-7E document to reflect the A-7D requirements.

  This became the **living document** of A-7D software behaviour.

• NRL continues to study the use of Tables in documenting software requirements and specifications (**SCR method**), including methodology and tool support.
Darlington Experience

Darlington nuclear shutdown system:
Two independent systems, each of which is responsible for shutting down the nuclear reaction in the event of an accident.

Project:
To determine whether the already-developed software and documentation met standards and could be certified.

Experience:
• Introduced program-function tables for documenting code
• Defined and executed a systematic inspection process
• 35-person-years task; relatively few important discrepancies found; but gained confidence in the code
A Program Function Table is an annotated Mixed Vector Table that describes the behaviour of a procedure or a sub-procedure.

### Procedure signature

\[
\text{NewDirection} \ (\text{dir}, \text{loc}, \text{Req})
\]

\[
R_1 = (\text{bottom} \leq \text{loc} \leq \text{top}) \Rightarrow \neg \text{ReqAbove(loc)} \land \text{ReqBelow(loc)} 
\]

### Vector Table

<table>
<thead>
<tr>
<th>dir’ =</th>
<th>‘dir</th>
<th>down</th>
<th>up</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dir</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Light[f]</th>
<th>(\forall f. \text{bottom} \leq f \leq \text{top})</th>
<th>Light[f] = <code>dir</code></th>
<th>(\forall f. \text{bottom} \leq f \leq \text{top})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light[f]</td>
<td>(\forall f. \text{bottom} \leq f \leq \text{top})</td>
<td>Light[f] = down</td>
<td>(\forall f. \text{bottom} \leq f \leq \text{top})</td>
</tr>
<tr>
<td>Light[f]</td>
<td>(\forall f. \text{bottom} \leq f \leq \text{top})</td>
<td>Light[f] = down</td>
<td>(\forall f. \text{bottom} \leq f \leq \text{top})</td>
</tr>
<tr>
<td>Light[f]</td>
<td>(\forall f. \text{bottom} \leq f \leq \text{top})</td>
<td>Light[f] = down</td>
<td>(\forall f. \text{bottom} \leq f \leq \text{top})</td>
</tr>
</tbody>
</table>

### Precondition

\[\neg \text{ReqAbove(loc)} \land \text{ReqBelow(loc)} \land \text{NC}(\text{Req,loc})\]

### Macros

- \(\text{ReqAbove}! = \exists f. \text{loc} < f \leq \text{top} \land \text{Req}[f]\)
- \(\text{ReqBelow}! = \exists f. \text{Bottom} \leq f < \text{loc} \land \text{Req}[f]\)

- Macro NoChange: \((\text{Req'} = \text{Req}) \land (\text{loc'} = \text{loc})\)
**Inspection Method**

**NewDirection (dir, loc, Req, Light)**

\[ R_1 = (\text{bottom} \leq \text{loc} \leq \text{top}) \Rightarrow \]

<table>
<thead>
<tr>
<th>dir' =</th>
<th>'dir</th>
<th>down</th>
<th>up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light'[f]</td>
<td>\forall f. \text{bottom} \leq f \leq \text{top} \land \text{Light}'[f] = 'dir</td>
<td>\forall f. \text{bottom} \leq f \leq \text{top} \land \text{Light}'[f] = \text{down}</td>
<td>\forall f. \text{bottom} \leq f \leq \text{top} \land \text{Light}'[f] = \text{down}</td>
</tr>
</tbody>
</table>

\(\land \text{NC(Req,loc)}\)

---

**Procedure NewDirection**

\[
\text{NewDirection (var \ direction:enum; var Light:Vector; floor:integer);} \\
\text{var I : integer;} \\
\text{begin} \\
\text{\hspace{1em}if \ PendingAbove(floor) <> PendingBelow(floor) then begin} \\
\text{\hspace{2em}if \ direction = up} \\
\text{\hspace{3em}then \ direction := down} \\
\text{\hspace{3em}else \ direction := up;} \\
\text{\hspace{2em}for \ i := \text{bottom} \ to \ \text{top} \ do} \\
\text{\hspace{3em}Light[i] := direction} \\
\text{\hspace{2em}end} \\
\text{end;} \\
\]

---

**PendingAbove(floor)**

| \(\exists f. [\text{floor} < f \leq \text{top} \land \text{Req}[f]] = \) | true | false |
| result' = | true | false |
| \(\land \text{NC(Req)}\) |

**PendingBelow(floor)**

| \(\exists f. [\text{bottom} \leq f < \text{floor} \land \text{Req}[f]] = \) | true | false |
| result' = | true | false |
| \(\land \text{NC(Req)}\) |
Systematic Inspections

Requirements

Design

Code

Program Fragment

Program Fragment

…

Program Fragment
Reviews and Inspections

① Well-formedness of tabular expressions
Reviews and Inspections

① Well-formedness of tabular expressions
① Requirements Validation

Domain Experts

Program

Function

Program

Fragment

Program

Fragment

Program

Fragment
Requirements Validation

Check that each case (subfunction, subrelation) produces the correct output.

\[
\begin{array}{cccc}
\text{dir} & \text{Up} & \text{Down} & \text{Down} & \text{Up} \\
\exists f.(f \geq \text{loc} \land \text{Req}[f]) & \text{true} & \text{false} & \text{---} & \text{true} \\
\exists f.(f \leq \text{loc} \land \text{Req}[f]) & \text{---} & \text{true} & \text{true} & \text{false}
\end{array}
\]
Reviews and Inspections

1. Well-formedness of tabular expressions
2. Requirements Validation
   2. Software Design Inspection

Domain Experts

Program Fragment

Program Fragment

Program Fragment
Reviews and Inspections

1. Well-formedness of tabular expressions
2. Requirements Validation
3. Software Design Inspection
4. Code Inspection

Program Fragment

function

program

function

Program Fragment

…”
Darlington Experience

Since then:
Ontario Hydro and the Atomic Energy Canada Limited (AECL) have developed a family of standards, procedures, and guidelines for developing safety critical software for use in nuclear power plants, incorporating

- **tabular, mathematical representations** of requirements, design, and code
- **systematic inspections** of requirements
- **mathematical verification** or **rigorous argument** that
  - the design meets the requirements
  - the code meets the design
Other Experiences

Experiences in which practitioners adopted the technology
• A-7E, A-7D aircraft (SCR)
• Ontario Hydro nuclear plant applications (Parnas Tables)
• Lockheed C130-J transport aircraft (CoRE)
• Medtronic medical applications (SCR)

Experiences that involved practitioners
• Trident (submarine) Emergency Preset System (SCR)
• AT&T Service Evaluation System (Parnas Tables)
• Traffic and Collision Avoidance System (RSML)
• Aircraft Separation Minima (HOL Parnas Tables)
• International Space Station (SCR)
Formal Semantics of Tables

Several Table types look alike, and readers may **misinterpret** a Table’s meaning when they are given only the Table’s informal, **ad hoc semantics**.

<table>
<thead>
<tr>
<th>dir</th>
<th>Up</th>
<th>Down</th>
<th>Down</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists f.(f \geq \text{loc} \land \text{Req}[f])$</td>
<td>true</td>
<td>false</td>
<td>---</td>
<td>true</td>
</tr>
<tr>
<td>$\exists f.(f \leq \text{loc} \land \text{Req}[f])$</td>
<td>---</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

---

**Decision Table** OR **Inverted Table**
Formal Semantics of Tables

To address this problem, there has been work on how to formulate the formal semantics of a tabular expression:

- **predicate rule** $p_T$ to define the expression’s **domain**
- **relation rule** $r_T$ to define the expression’s **range**
- **composition rule** $C_T$ to define how to combine subexpressions

---

<table>
<thead>
<tr>
<th>Decision Table</th>
<th>Inverted Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_T$: $H_2 = G$</td>
<td>$p_T$: $H_2 \land G$</td>
</tr>
<tr>
<td>$r_T$: $H_3$</td>
<td>$r_T$: $H_3$</td>
</tr>
<tr>
<td>$C_T$: $\bigcup_{j=1}^{4} \bigotimes_{i=1}^{4} F_{i,j}$</td>
<td>$C_T$: $\bigcup_{i=1}^{3} \left( \bigcup_{j=1}^{3} F_{i,j} \right)$</td>
</tr>
</tbody>
</table>
Table Transformations

One may want to **transform** one table to another representation, to formulate a more **compact expression** or determine the equivalence of two table expressions.

<table>
<thead>
<tr>
<th>dir=Up</th>
<th>dir=Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists f.(f \geq loc \land Req[f]) )</td>
<td>( \neg \exists f.(f \geq loc \land Req[f]) \land \exists f.(f &lt; loc \land Req[f]) )</td>
</tr>
<tr>
<td>( \neg \exists f.(f \leq loc \land Req[f]) \land \exists f.(f &gt; loc \land Req[f]) )</td>
<td>( \exists f.(f \leq loc \land Req[f]) )</td>
</tr>
</tbody>
</table>

| \( \exists f.(f \leq loc \land Req[f]) \) | \( true \) | \( false \) |

| \( \exists f.(f \leq loc \land Req[f]) \) | \( true \) | \( false \) |

<table>
<thead>
<tr>
<th>dir</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>dir</td>
</tr>
</tbody>
</table>
Table Transformations

But even a simple transformations, like one that exchanges grid elements with header elements, can require reorganization and simplification to produce a concise table.
Automated Checking

Significant human effort may be needed to check that a table is consistent and that it covers the expression’s domain. Since these checks are application-independent and can be expressed as constraints on predicates, many can be automated.

<table>
<thead>
<tr>
<th>Req[loc]</th>
<th>¬Req[loc]</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬∃f.Req[f]</td>
<td>∃f.(f&gt;loc ∧ Req[f]) ∧ ¬∃f.(f&lt;loc ∧ Req[f])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dir'</th>
<th>dir</th>
<th>dir</th>
<th>Up</th>
<th>Down</th>
<th>dir</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>idle</td>
<td>moving</td>
<td>moving</td>
<td>moving</td>
<td></td>
</tr>
</tbody>
</table>
Reasoning about Table Composition

Each Table documents a separate concern. If the concerns are not completely separate (e.g., if they react to changes in the same variables) then, we need to review their composition.

Application-Independent
- reachability
- deadlock
- cycle detection

Application-Dependent
- abstractions
- coordination
- safety properties
- liveness properties
- invariant generation
Summary

Parnas Tables are practical formalisms that

- emphasize **abstraction** and **separation of concerns**
- are amenable to **readable, write-able, and review-able** yet precise software documents
- are **useful at different degrees of formalism**

| Tabular expressions | Tabular expressions of mathematical relations | Systematic inspections | Inspections of table compositions | Mathematical verification |