A Framework for Supporting Updates to Database Components

Stephen J. Hegner
Umeå University
Department of Computing Science
Sweden

The Composition of this Talk

Part 1: The model of database components.

> **Based upon**: A Model of Database Components and their Interconnection Based upon Communicating Views (EJC 2007)

Part 2: Basic update support via cooperation.

> **Based upon**: Update Support for Database Views via Cooperation (ADBIS 2007, with Peggy Schmidt)

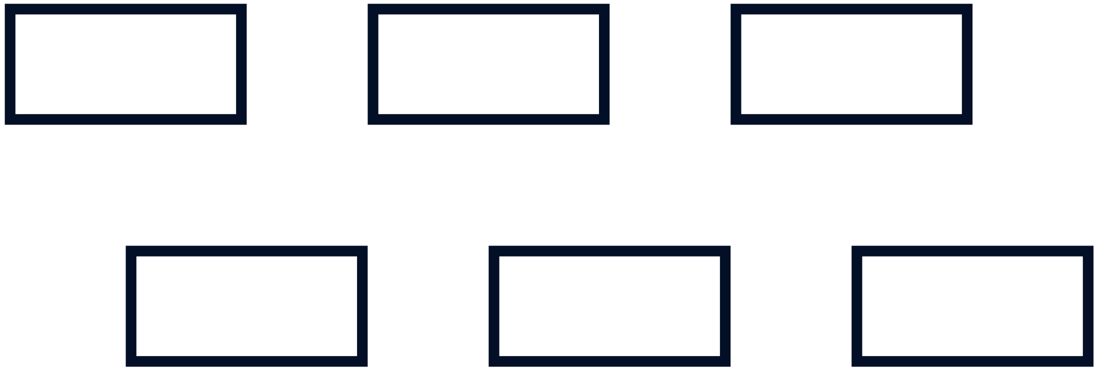
Part 3: Current work on update support.

> **Based upon**: Updates to Database Schemata Defined by Interconnected Components (TR 0805 with Peggy Schmidt, to appear).

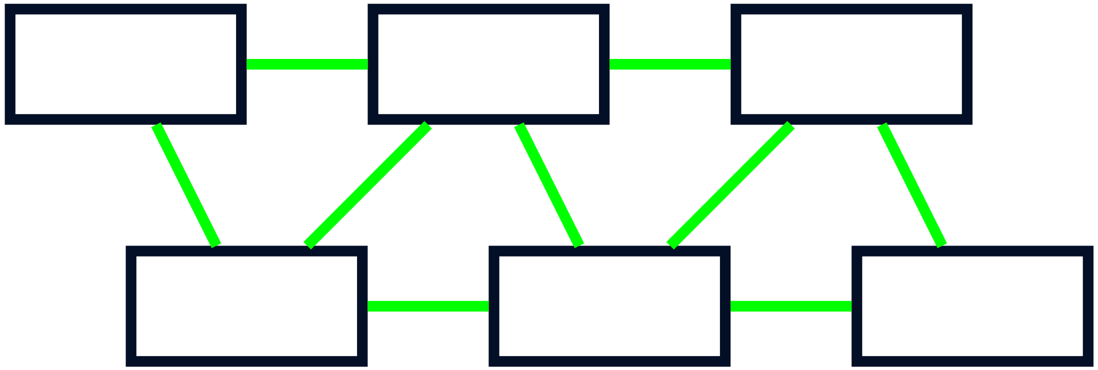
The Model of Database Components

Managing the Complexity of Large-Scale Systems

- It has long been known that the complexity of large systems may be managed by modelling them as components ...

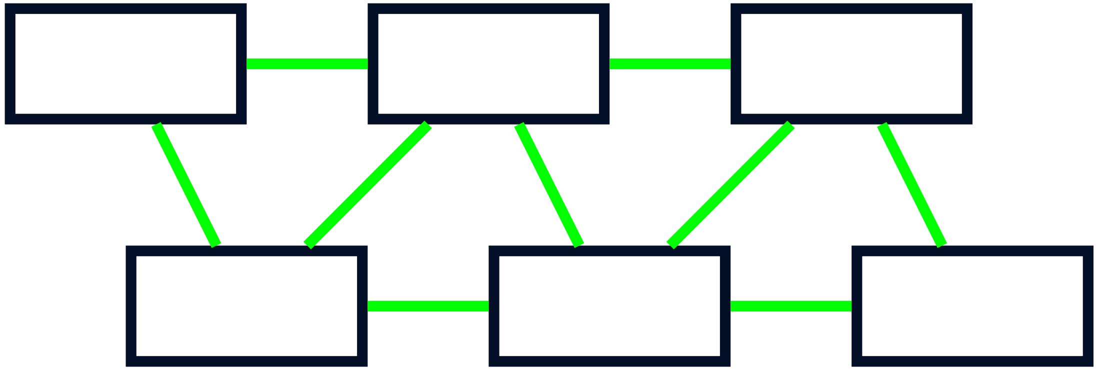


- It has long been known that the complexity of large systems may be managed by modelling them as components ...
- ... together with their interconnections.



Managing the Complexity of Large-Scale Systems

- It has long been known that the complexity of large systems may be managed by modelling them as components …
- … together with their interconnections.

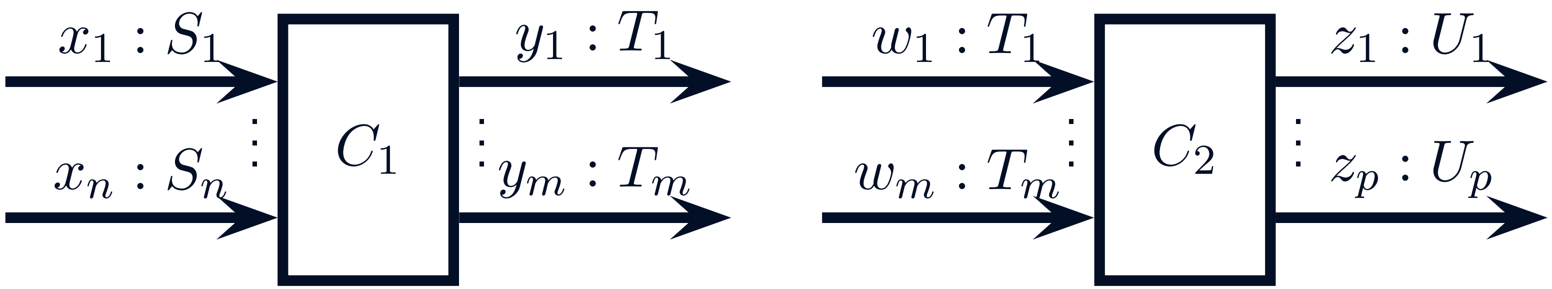


- This approach is standard for hardware systems.
- It is far less common for software systems.
- A reason is perhaps that software components are far less standard than hardware components.

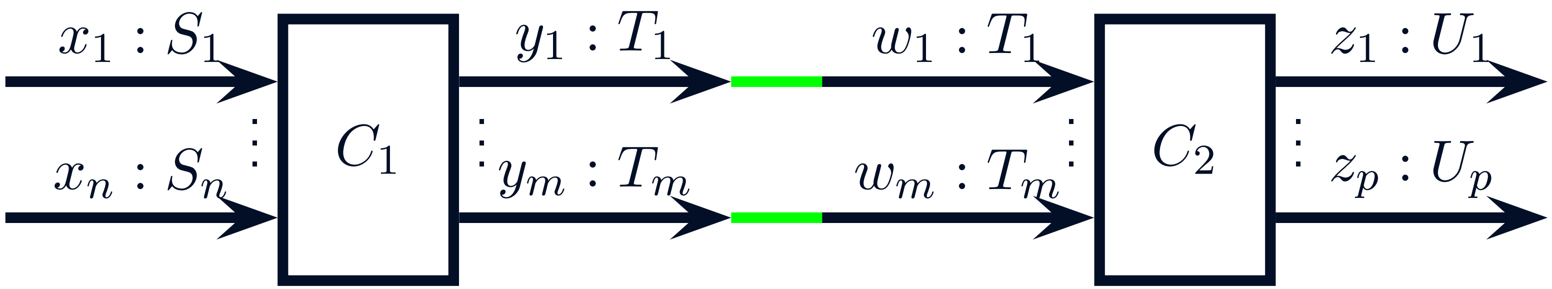
- Over the past decade, Manfred Broy has developed a fairly comprehensive theory of software components based upon *state machines*.

A Model of Software and Database Components

- Over the past decade, Manfred Broy has developed a fairly comprehensive theory of software components based upon *state machines*.
- The input and output connectors are called *channels*.
- Data on the channels takes the form of *streams*.



- Over the past decade, Manfred Broy has developed a fairly comprehensive theory of software components based upon *state machines*.
- The input and output connectors are called *channels*.
- Data on the channels takes the form of *streams*.
- The S_i's and T_i's are the *types* of the channels.
- Interconnection must respect the channel types.



Co-Design and Database Components

- Bernhard Thalheim has developed a corresponding model of database components.

Co-Design and Database Components

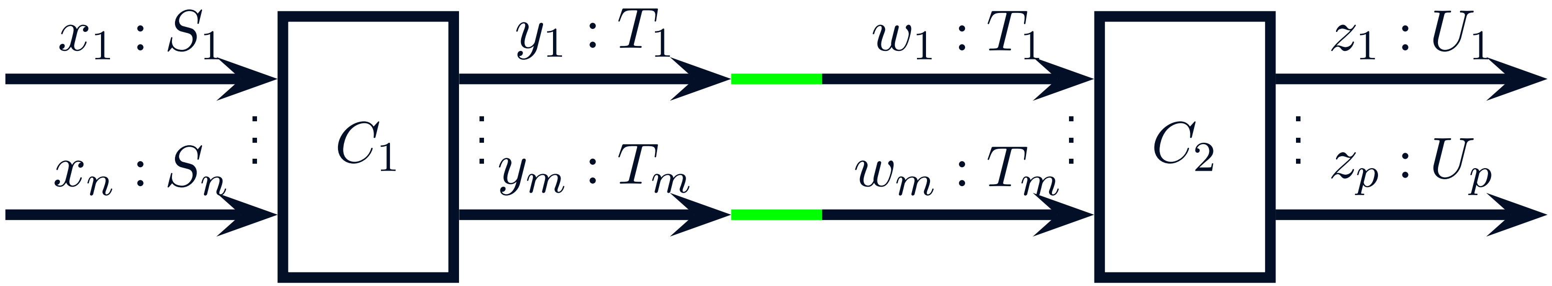
- Bernhard Thalheim has developed a corresponding model of database components.
- The channels are database views.

Co-Design and Database Components

- Bernhard Thalheim has developed a corresponding model of database components.
- The channels are database views.
- The principle upon which this model is based in *co-design* – that applications are integrated into the components during the design process.

Co-Design and Database Components

- Bernhard Thalheim has developed a corresponding model of database components.
- The channels are database views.
- The principle upon which this model is based in *co-design* – that applications are integrated into the components during the design process.
- This implies that communication via streams of view states as the fundamental notion of communication between database components.



An Alternative Model of Database Components for this Work

- Develop a theory of database components which is grounded in classical database constructs.

An Alternative Model of Database Components for this Work

- Develop a theory of database components which is grounded in classical database constructs.
- Notions such as schemata and views should be the cornerstones of the constructions.

An Alternative Model of Database Components for this Work

- Develop a theory of database components which is grounded in classical database constructs.
- Notions such as schemata and views should be the cornerstones of the constructions.
- It is assumed that models of communication will vary from application to application. Therefore ...

An Alternative Model of Database Components for this Work

- Develop a theory of database components which is grounded in classical database constructs.
- Notions such as schemata and views should be the cornerstones of the constructions.
- It is assumed that models of communication will vary from application to application. Therefore ...
- An explicit model of communication is not part of the fundamental model, but rather something which is to be added as needed.

An Alternative Model of Database Components for this Work

- Develop a theory of database components which is grounded in classical database constructs.
- Notions such as schemata and views should be the cornerstones of the constructions.
- It is assumed that models of communication will vary from application to application. Therefore ...
- An explicit model of communication is not part of the fundamental model, but rather something which is to be added as needed.
- The theory should be flexible and not limited to a particular data model.

An Alternative Model of Database Components for this Work

- Develop a theory of database components which is grounded in classical database constructs.
- Notions such as schemata and views should be the cornerstones of the constructions.
- It is assumed that models of communication will vary from application to application. Therefore ...
- An explicit model of communication is not part of the fundamental model, but rather something which is to be added as needed.
- The theory should be flexible and not limited to a particular data model.
- Nevertheless, it should be firmly motivated by constructions in the classical relational model.

$$\begin{tikzcd} & (R[ABC], \{A \to B, B \to C\}) \arrow[dl, "\pi_{AB}"'] \arrow[dr, "\pi_{BC}"] & \\ (R[AB], \{A \to B\}) & & (R[BC], \{B \to C\}) \end{tikzcd}$$

- Consider the lossless and dependency-preserving decomposition to the left.

Classical Relational Decomposition Revisited

$$
\begin{tikzcd}
 & (R[ABC], \{A \to B, B \to C\}) \arrow[dl, "\pi_{AB}"'] \arrow[dr, "\pi_{BC}"] & \\
(R[AB], \{A \to B\}) \arrow[dr, "\pi_B"'] & & (R[BC], \{B \to C\}) \arrow[dl, "\pi_B"] \\
 & (R[B], \emptyset) &
\end{tikzcd}
$$

- Consider the lossless and dependency-preserving decomposition to the left.
- The decomposed databases are precisely those which agree on their B-projections (*independence*: Rissanen 1977).

$$
\begin{tikzcd}[column sep=tiny]
 & (R[ABC], \{A \to C, B \to C\}) \arrow[dl, "\pi_{AB}"'] \arrow[dr, "\pi_{BC}"] & \\
(R[AB], \emptyset) \arrow[dr, "\pi_B"'] & & (R[BC], \{B \to C\}) \arrow[dl, "\pi_B"] \\
 & (R[B], \emptyset) &
\end{tikzcd}
$$

- Consider the lossless and dependency-preserving decomposition to the left.
- The decomposed databases are precisely those which agree on their B-projections (*independence*: Rissanen 1977).
- Note that this construction does not work if the decomposition is not dependency preserving.
- It is impossible to enforce $A \to C$ with such a construction.

Classical Relational Decomposition Revisited

$$
\begin{tikzcd}
 & (R[ABC], \{A \to C, B \to C\}) \arrow[dl, "\pi_{AB}"'] \arrow[dr, "\pi_{BC}"] & \\
(R[AB], \emptyset) \arrow[dr, "\pi_B"'] & & (R[BC], \{B \to C\}) \arrow[dl, "\pi_B"] \\
 & (R[B], \emptyset) &
\end{tikzcd}
$$

- Consider the lossless and dependency-preserving decomposition to the left.
- The decomposed databases are precisely those which agree on their B-projections (*independence*: Rissanen 1977).
- Note that this construction does not work if the decomposition is not dependency preserving.
- It is impossible to enforce $A \to C$ with such a construction.

- The property of being *dependency preserving* (*commuting congruences*) is thus critical.

Classical Relational Decomposition Revisited

$$\begin{tikzcd}[column sep=tiny]
 & (R[ABC], \{A \to B, B \to C\}) \arrow[dl, "\pi_{AB}"'] \arrow[dr, "\pi_{BC}"] & \\
(R[AB], \{A \to B\}) \arrow[dr, "\pi_B"'] & & (R[BC], \{B \to C\}) \arrow[dl, "\pi_B"] \\
 & (R[B], \emptyset) &
\end{tikzcd}$$

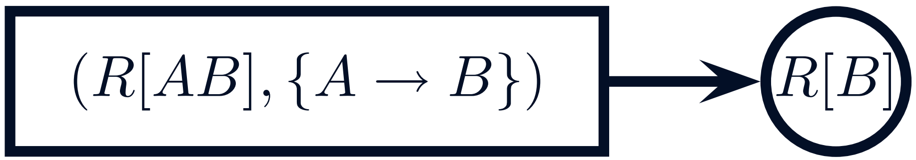
- Consider the lossless and dependency-preserving decomposition to the left.
- The decomposed databases are precisely those which agree on their B-projections (*independence*: Rissanen 1977).
- Note that this construction does not work if the decomposition is not dependency preserving.
- It is impossible to enforce $A \to C$ with such a construction.

- The property of being *dependency preserving* (*commuting congruences*) is thus critical.
- Goal: Use these ideas to motivate the definition of a component.

The Components of Classical Relational Decomposition

- A *database component* is a schema together with a set of views.

- A *database component* is a schema together with a set of views.
- Define the component $K_{AB} = ((R[AB], \{A \rightarrow B\}), \{\Pi_B^{R[AB]}\})$
  with the view $\Pi_B^{R[AB]} = ((R[B], \emptyset), \pi_B)$.

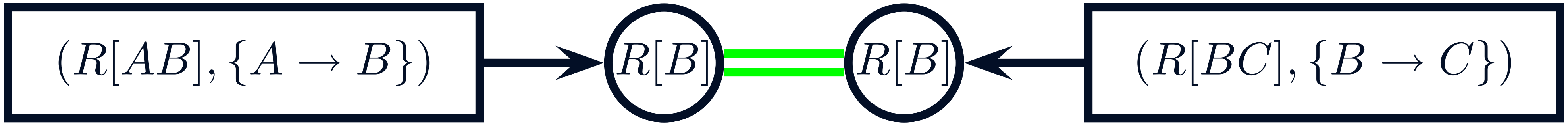


- A *database component* is a schema together with a set of views.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  with the view $\Pi_B^{R[AB]} = ((R[B], \emptyset), \pi_B)$.
- Define the component $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ similarly.

$$\boxed{(R[AB], \{A \to B\})} \longrightarrow R[B] \qquad\qquad R[B] \longleftarrow \boxed{(R[BC], \{B \to C\})}$$

The Components of Classical Relational Decomposition

- A *database component* is a schema together with a set of views.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  with the view $\Pi_B^{R[AB]} = ((R[B], \emptyset), \pi_B)$.
- Define the component $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ similarly.
- Connecting the ports of these two components results in a combination which is isomorphic to $(R[ABC], \{A \to B, B \to C\})$.



The Components of Classical Relational Decomposition

- A *database component* is a schema together with a set of views.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  with the view $\Pi_B^{R[AB]} = ((R[B], \emptyset), \pi_B)$.
- Define the component $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ similarly.
- Connecting the ports of these two components results in a combination which is isomorphic to $(R[ABC], \{A \to B, B \to C\})$.

$$\boxed{(R[AB], \{A \to B\})} \longrightarrow R[B] \longleftarrow \boxed{(R[BC], \{B \to C\})}$$

The Components of Classical Relational Decomposition

- A *database component* is a schema together with a set of views.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  with the view $\Pi_B^{R[AB]} = ((R[B], \emptyset), \pi_B)$.
- Define the component $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ similarly.
- Connecting the ports of these two components results in a combination which is isomorphic to $(R[ABC], \{A \to B, B \to C\})$.

$$\boxed{(R[AB], \{A \to B\})} \longrightarrow R[B] \longleftarrow \boxed{(R[BC], \{B \to C\})}$$

- This recaptures lossless and dependency-preserving decomposition, but as a *composition* rather than as a decomposition.

The Components of Classical Relational Decomposition

- A *database component* is a schema together with a set of views.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  with the view $\Pi_B^{R[AB]} = ((R[B], \emptyset), \pi_B)$.
- Define the component $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ similarly.
- Connecting the ports of these two components results in a combination which is isomorphic to $(R[ABC], \{A \to B, B \to C\})$.

- This recaptures lossless and dependency-preserving decomposition, but as a *composition* rather than as a decomposition.
- No reference to the main schema is necessary.

- A *database schema* $\mathbf{D}$ is modelled by its set $\mathrm{LDB}(\mathbf{D})$ of underlying states.

General Notions of Database Components

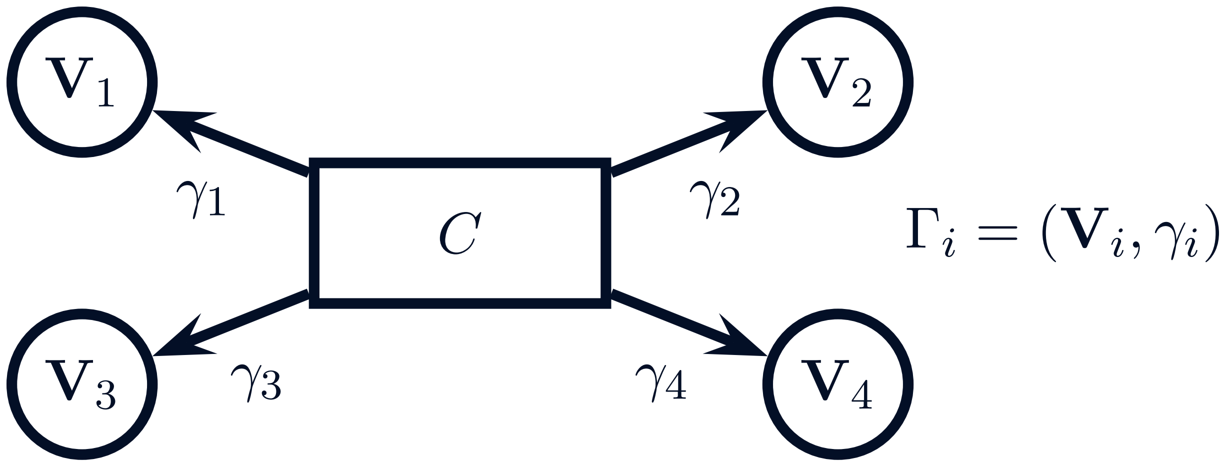
- A *database schema* $\mathbf{D}$ is modelled by its set $\mathsf{LDB}(\mathbf{D})$ of underlying states.
- A *view* of $\mathbf{D}$ is a pair $\Gamma = (\mathbf{V}, \gamma)$ in which:
  - $\mathbf{V}$ is a database schema;
  - $\gamma : \mathbf{D} \to \mathbf{V}$ is a database morphism whose underlying function $\mathring{\gamma} : \mathsf{LDB}(\mathbf{D}) \to \mathsf{LDB}(\mathbf{V})$ is surjective.

General Notions of Database Components

- A *database schema* $\mathbf{D}$ is modelled by its set $\mathsf{LDB}(\mathbf{D})$ of underlying states.
- A *view* of $\mathbf{D}$ is a pair $\Gamma = (\mathbf{V}, \gamma)$ in which:
  - $\mathbf{V}$ is a database schema;
  - $\gamma : \mathbf{D} \to \mathbf{V}$ is a database morphism whose underlying function $\mathring{\gamma} : \mathsf{LDB}(\mathbf{D}) \to \mathsf{LDB}(\mathbf{V})$ is surjective.
- A *database component* is a pair $C = (\mathsf{Schema}(C), \mathsf{Ports}(C))$ in which:
  - $\mathsf{Schema}(C)$ is a database schema.
  - $\mathsf{Ports}(C)$ is a finite set of views of $\mathsf{Schema}(C)$, called the *ports* of C.

General Notions of Database Components

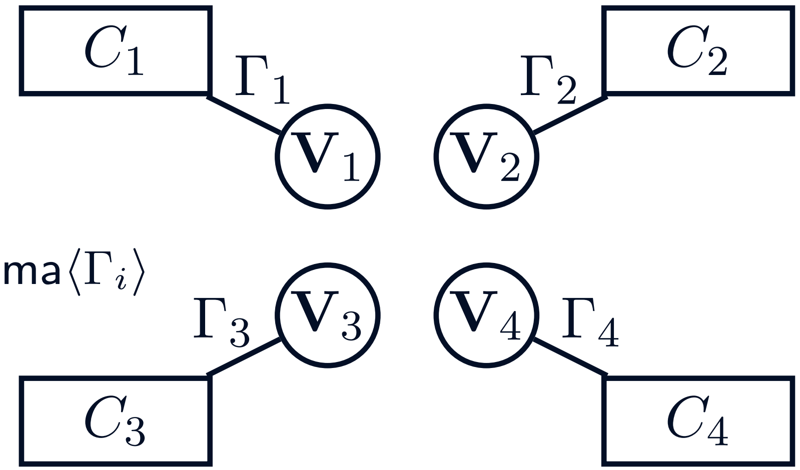
- A *database schema* $\mathbf{D}$ is modelled by its set $\mathsf{LDB}(\mathbf{D})$ of underlying states.
- A *view* of $\mathbf{D}$ is a pair $\Gamma = (\mathbf{V}, \gamma)$ in which:
  - $\mathbf{V}$ is a database schema;
  - $\gamma : \mathbf{D} \to \mathbf{V}$ is a database morphism whose underlying function $\mathring{\gamma} : \mathsf{LDB}(\mathbf{D}) \to \mathsf{LDB}(\mathbf{V})$ is surjective.
- A *database component* is a pair $C = (\mathsf{Schema}(C), \mathsf{Ports}(C))$ in which:
  - $\mathsf{Schema}(C)$ is a database schema.
  - $\mathsf{Ports}(C)$ is a finite set of views of $\mathsf{Schema}(C)$, called the *ports* of C.



Star Compatibility — the Condition for Interconnection

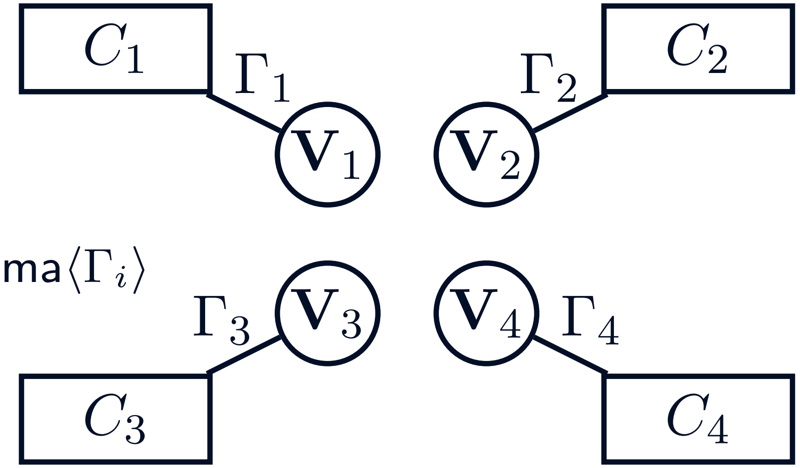
- A single interconnection of components is called a *star configuration*.

$\mathbf{V}_i \stackrel{\text{def}}{=} \mathsf{Schema}\langle\Gamma_i\rangle$



Star Compatibility — the Condition for Interconnection

- A single interconnection of components is called a *star configuration*.
- The ports of the connected components must have **identical** port schemata.

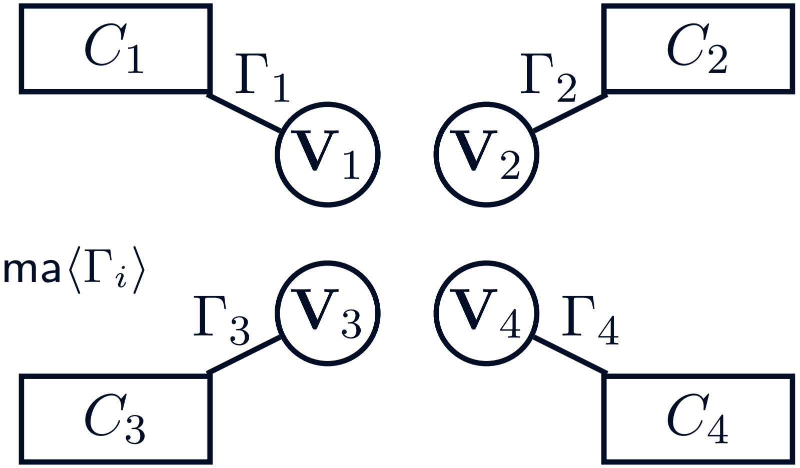


$$\mathbf{V}_i \stackrel{\text{def}}{=} \text{Schema}\langle \Gamma_i \rangle$$

$$\mathbf{V}_1 = \mathbf{V}_2 = \mathbf{V}_3 = \mathbf{V}_4 = \mathbf{V} \stackrel{\text{def}}{=} \text{Schema}\langle \Gamma \rangle$$

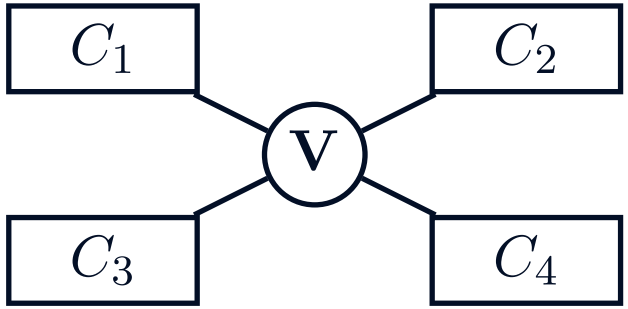
Star Compatibility — the Condition for Interconnection

- A single interconnection of components is called a *star configuration*.
- The ports of the connected components must have **identical** port schemata.
- This condition is called *star compatibility*.



$\mathbf{V}_i \stackrel{\text{def}}{=} \text{Schema}\langle \Gamma_i \rangle$

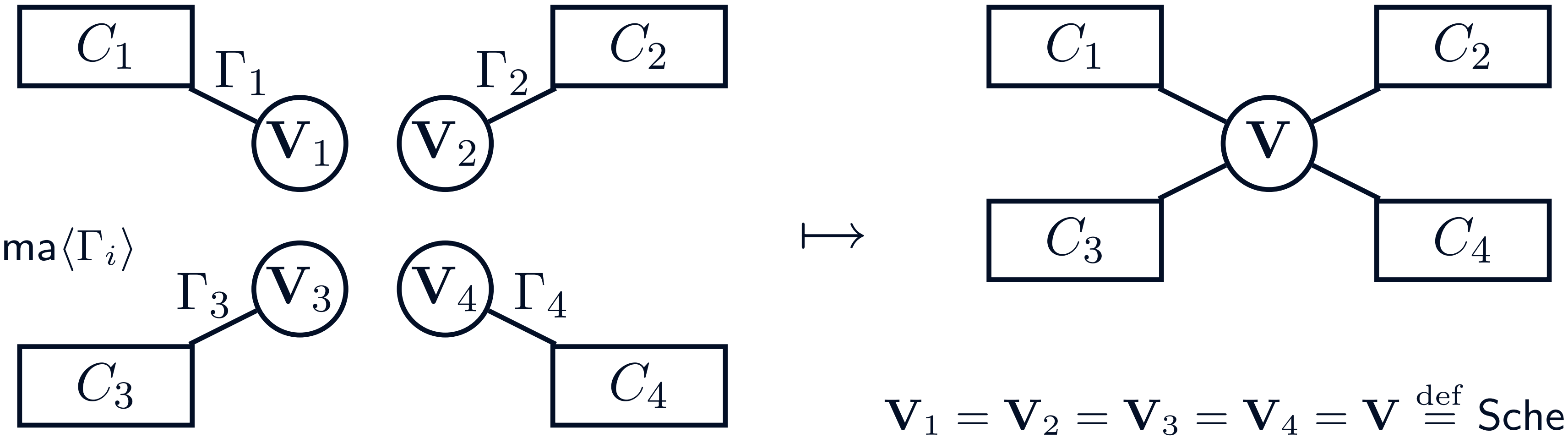
$\longmapsto$



$\mathbf{V}_1 = \mathbf{V}_2 = \mathbf{V}_3 = \mathbf{V}_4 = \mathbf{V} \stackrel{\text{def}}{=} \text{Schema}\langle \Gamma \rangle$

Star Compatibility — the Condition for Interconnection

- A single interconnection of components is called a *star configuration*.
- The ports of the connected components must have **identical** port schemata.
- This condition is called *star compatibility*.

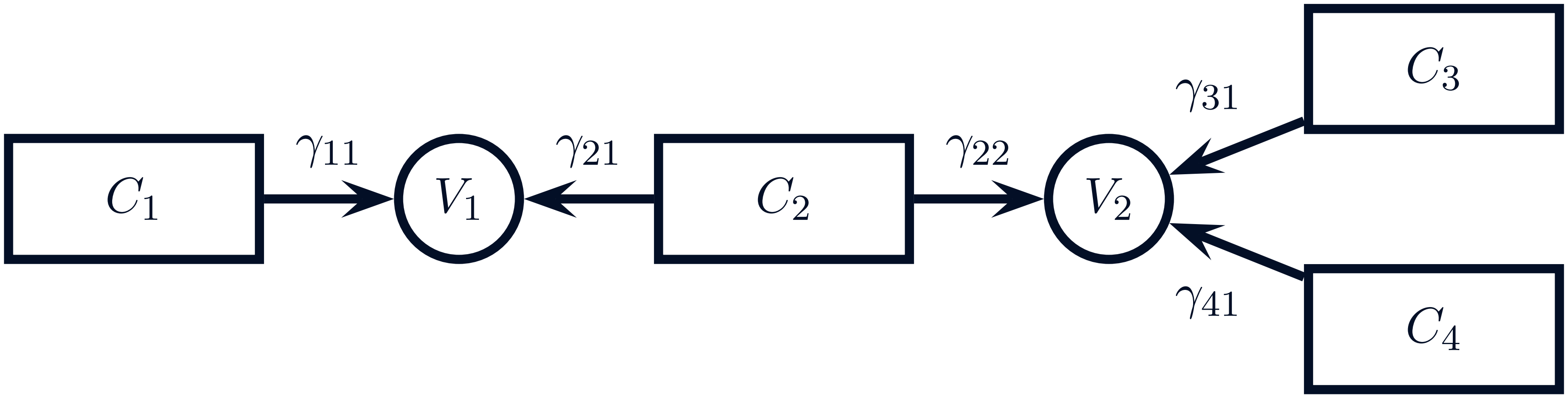


$\mathbf{V}_i \stackrel{\text{def}}{=} \mathsf{Schema}\langle \Gamma_i \rangle$

$\mathbf{V}_1 = \mathbf{V}_2 = \mathbf{V}_3 = \mathbf{V}_4 = \mathbf{V} \stackrel{\text{def}}{=} \mathsf{Schema}\langle \Gamma \rangle$

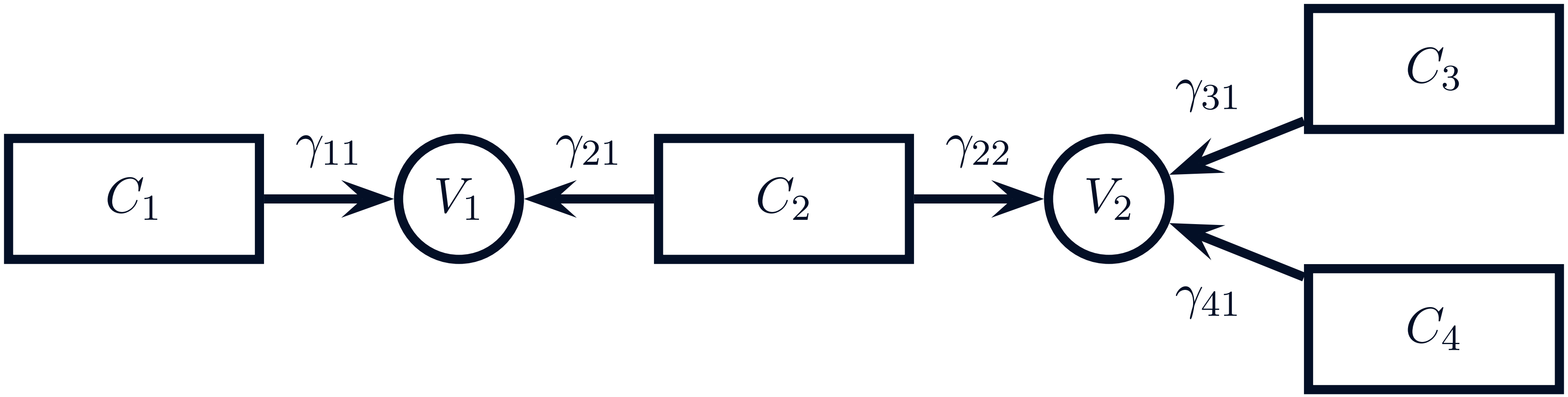
- An *interconnection family* is a set of star interconnections.

- In the case of an acyclic interconnection, the state is very easy to describe.



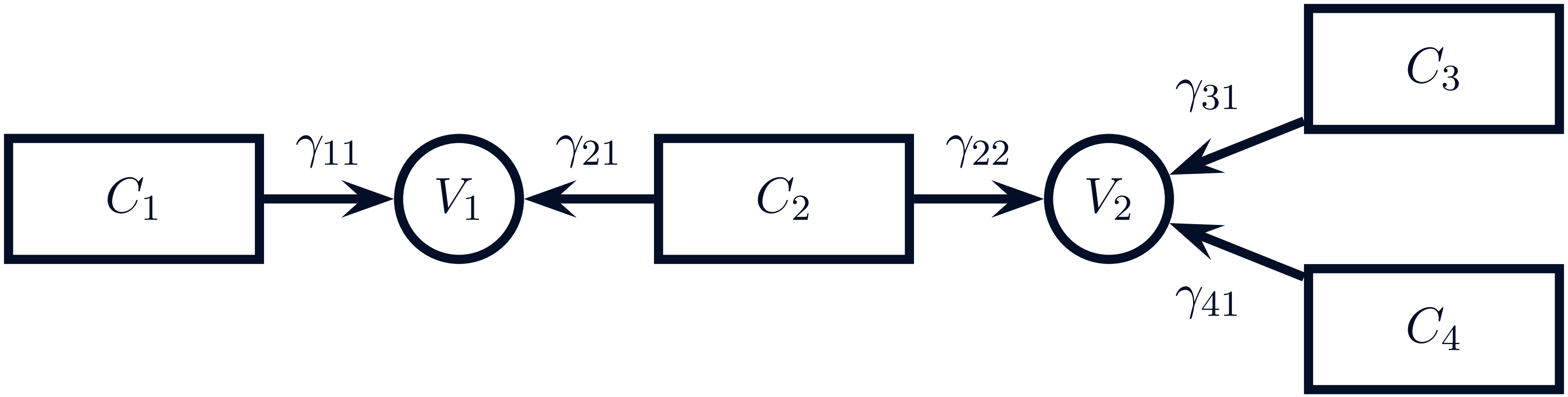
The State of an Interconnected Set of Components

- In the case of an acyclic interconnection, the state is very easy to describe.
- $$\mathsf{LDB}(\text{interconnection}) = \{(M_1, M_2, M_3, M_4) \in \mathsf{LDB}(C_1) \times \mathsf{LDB}(C_2) \times \mathsf{LDB}(C_3) \times \mathsf{LDB}(C_4)\} \mid \mathring{\gamma}_{11}(M_1) = \mathring{\gamma}_{21}(M_2) \;\wedge\; \mathring{\gamma}_{22}(M_2) = \mathring{\gamma}_{31}(M_3) = \mathring{\gamma}_{41}(M_4) \}$$

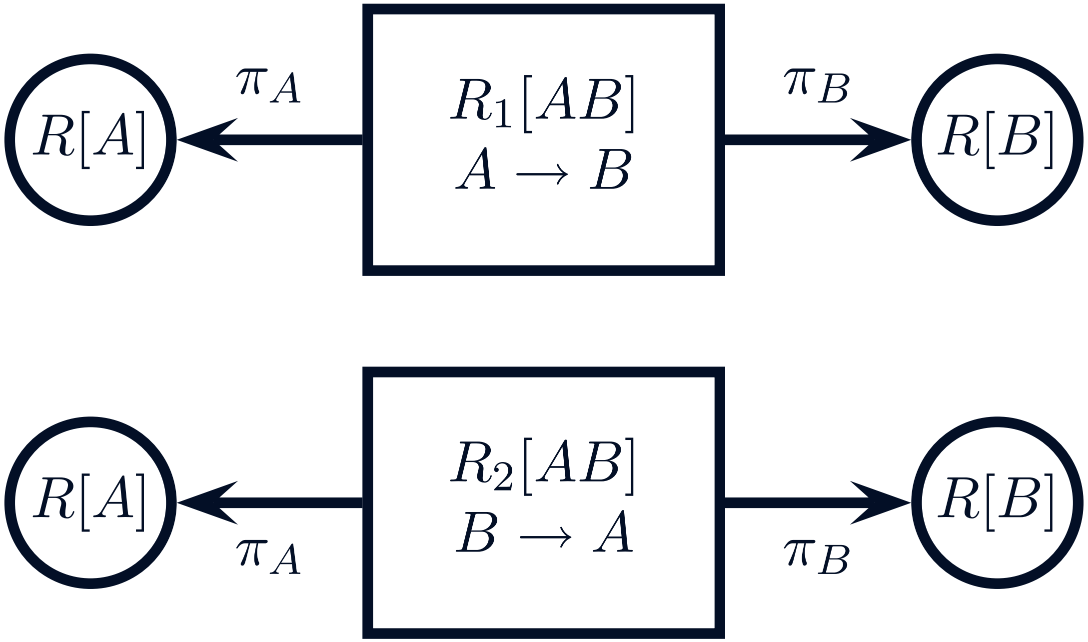


The State of an Interconnected Set of Components

- In the case of an acyclic interconnection, the state is very easy to describe.
- $$\begin{aligned}&\mathsf{LDB}(\text{interconnection}) = \\ &\quad \{(M_1, M_2, M_3, M_4) \in \mathsf{LDB}(C_1) \times \mathsf{LDB}(C_2) \times \mathsf{LDB}(C_3) \times \mathsf{LDB}(C_4)\} \\ &\qquad \mid\ \mathring{\gamma}_{11}(M_1) = \mathring{\gamma}_{21}(M_2) \ \wedge\ \mathring{\gamma}_{22}(M_2) = \mathring{\gamma}_{31}(M_3) = \mathring{\gamma}_{41}(M_4)\ \}\end{aligned}$$
- Unfortunately, serious complications arise if the interconnection is allowed to be cyclic.

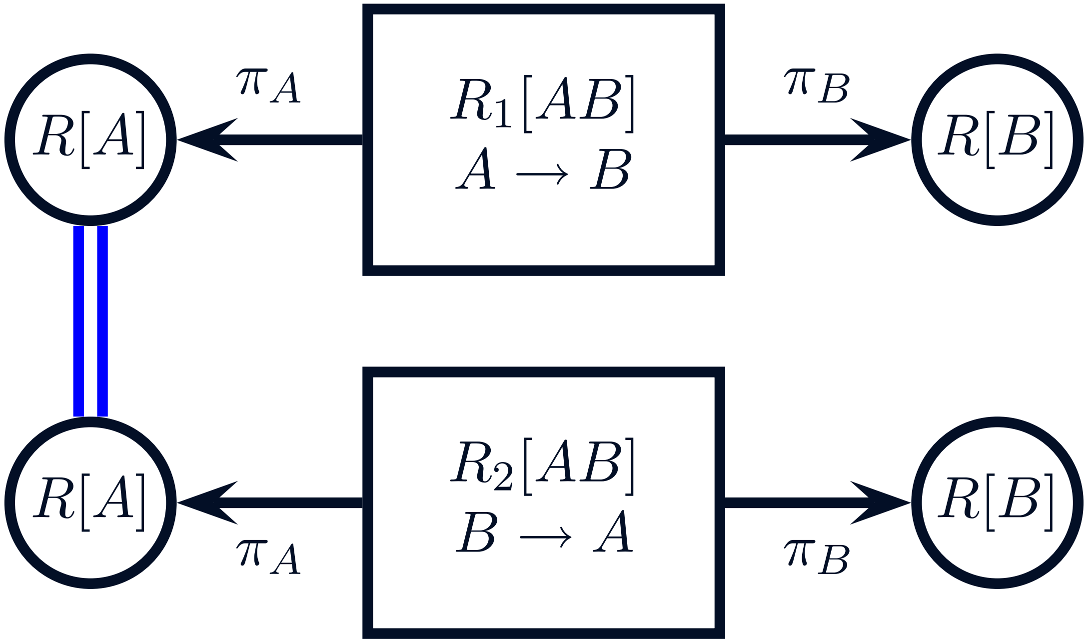


Cyclicity Issues

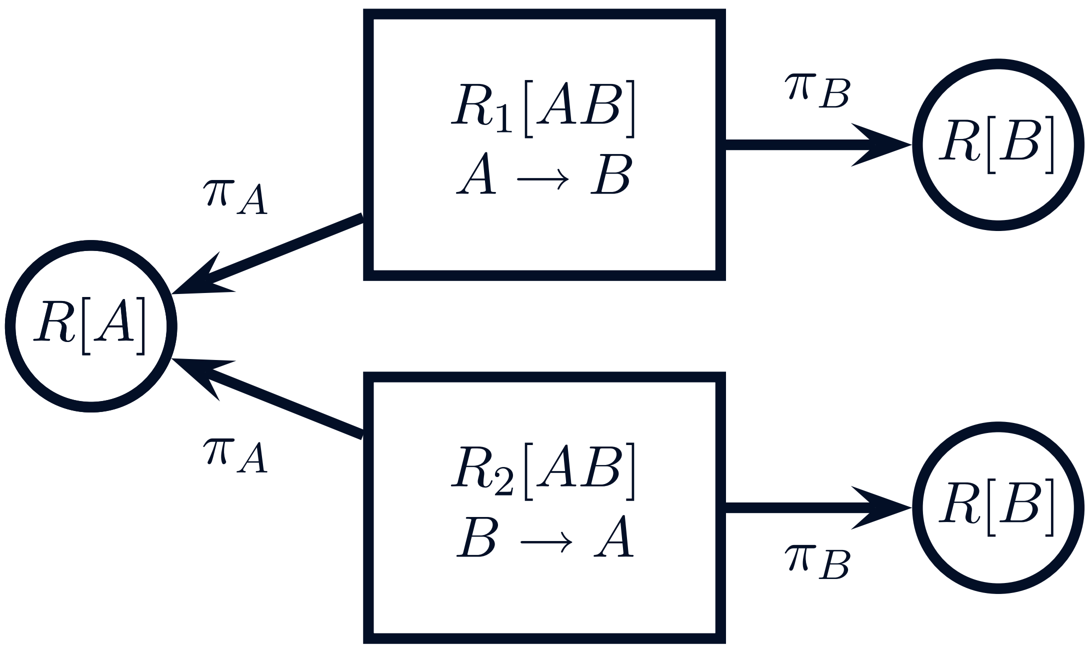


- Assume that all databases are finite.

Cyclicity Issues

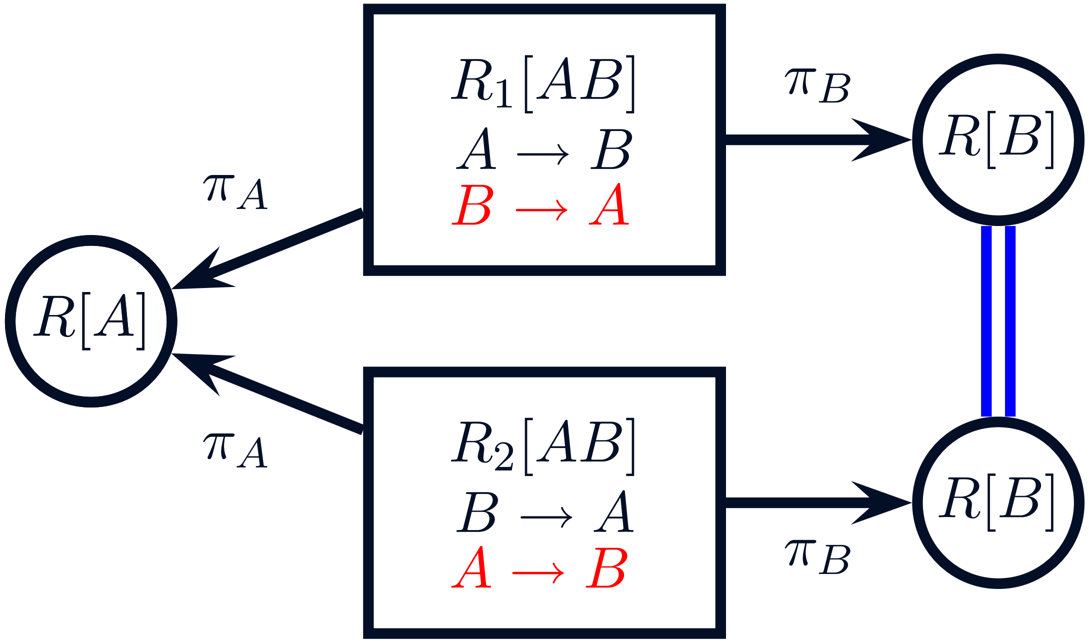


- Assume that all databases are finite.
- Connecting one set of ports causes no problems.

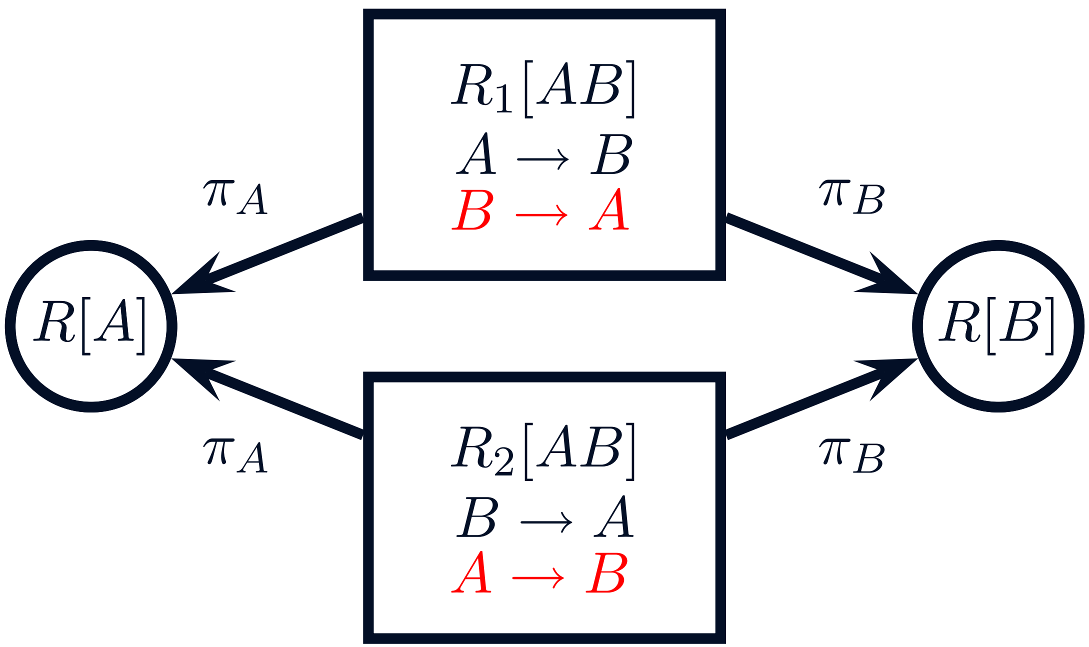


- Assume that all databases are finite.
- Connecting one set of ports causes no problems.

Cyclicity Issues

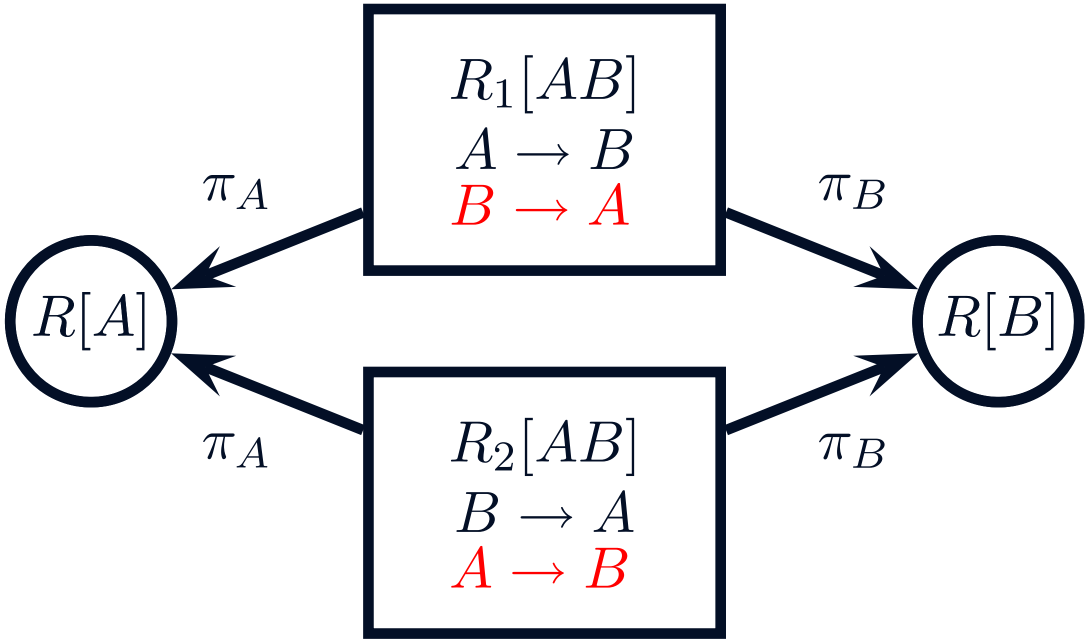


- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

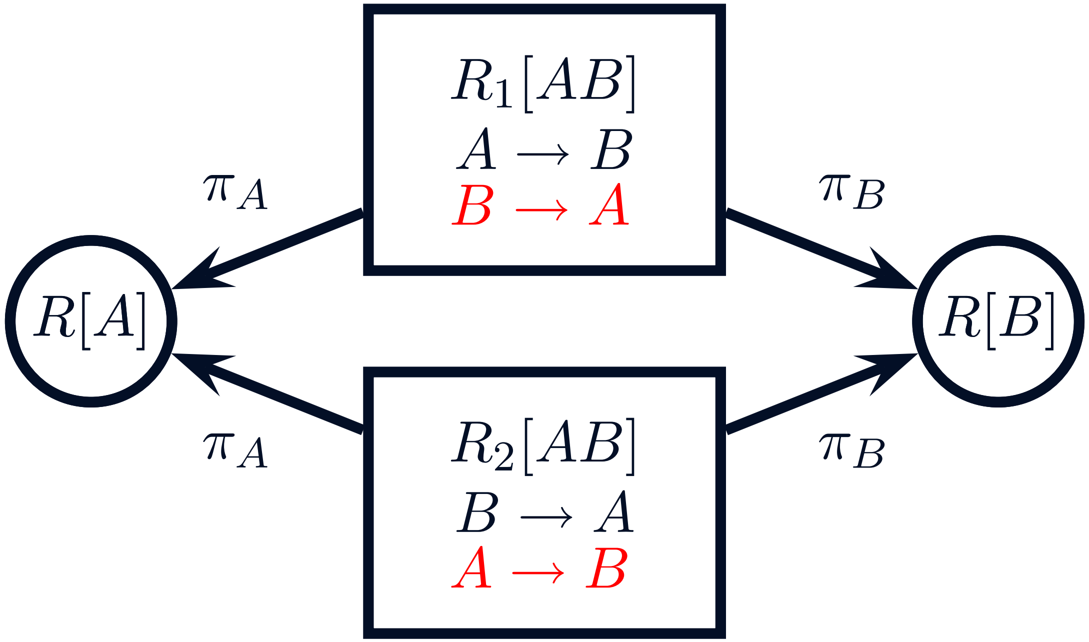
Cyclicity Issues



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

- Good news: the schemata of the ports have not changed.
- The ports are still views, in the sense that the underlying mappings are still surjective.

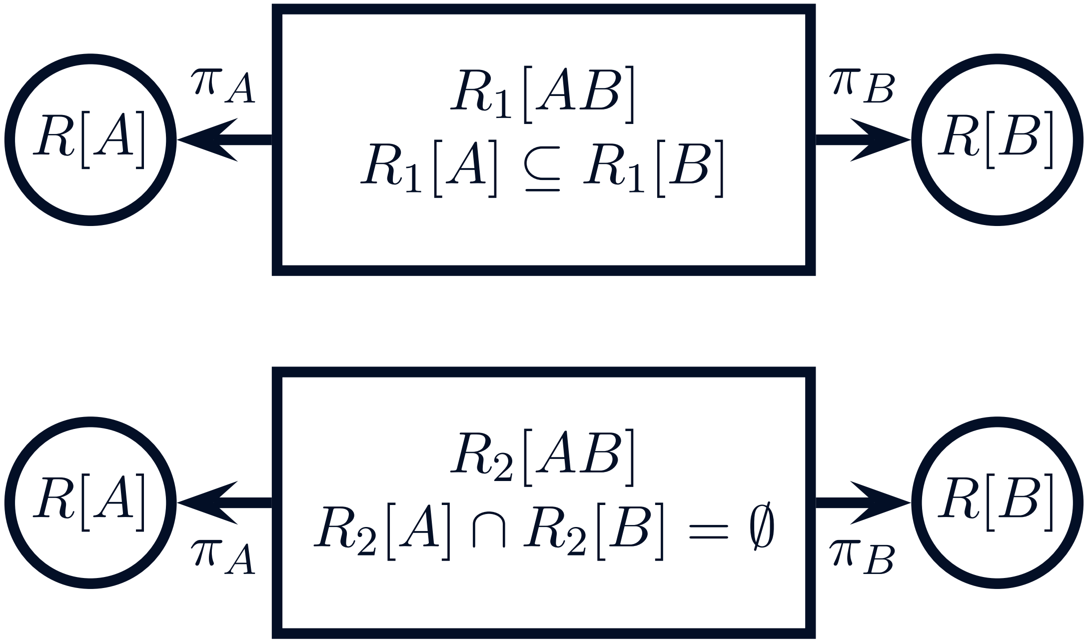
Cyclicity Issues



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

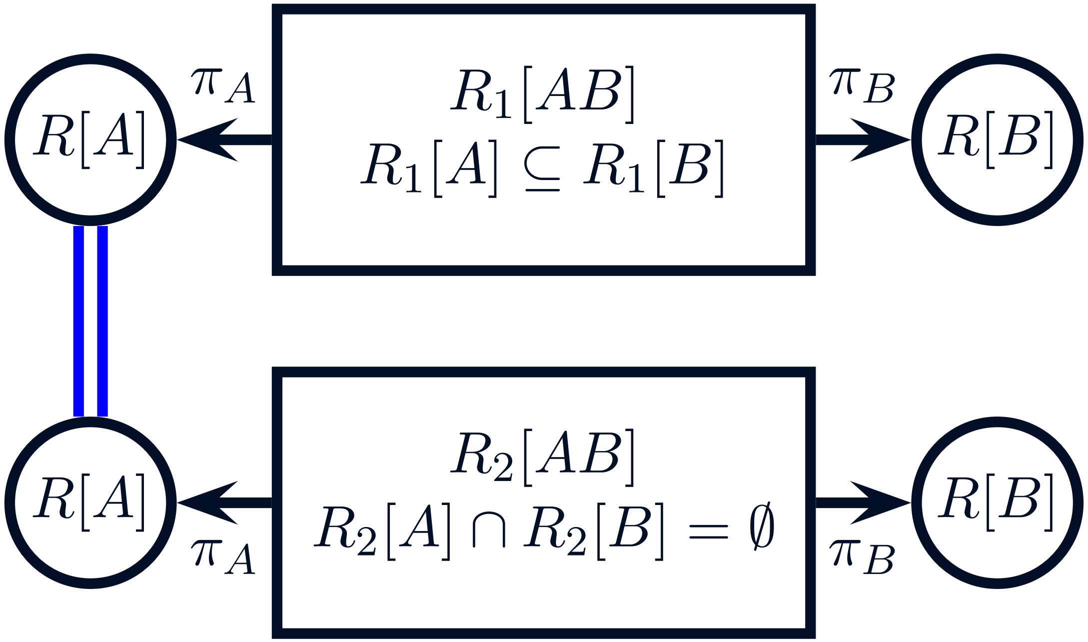
- Good news: the schemata of the ports have not changed.
- The ports are still views, in the sense that the underlying mappings are still surjective.
- This type of cyclic interconnection is manageable.

Cyclicity Issues 2



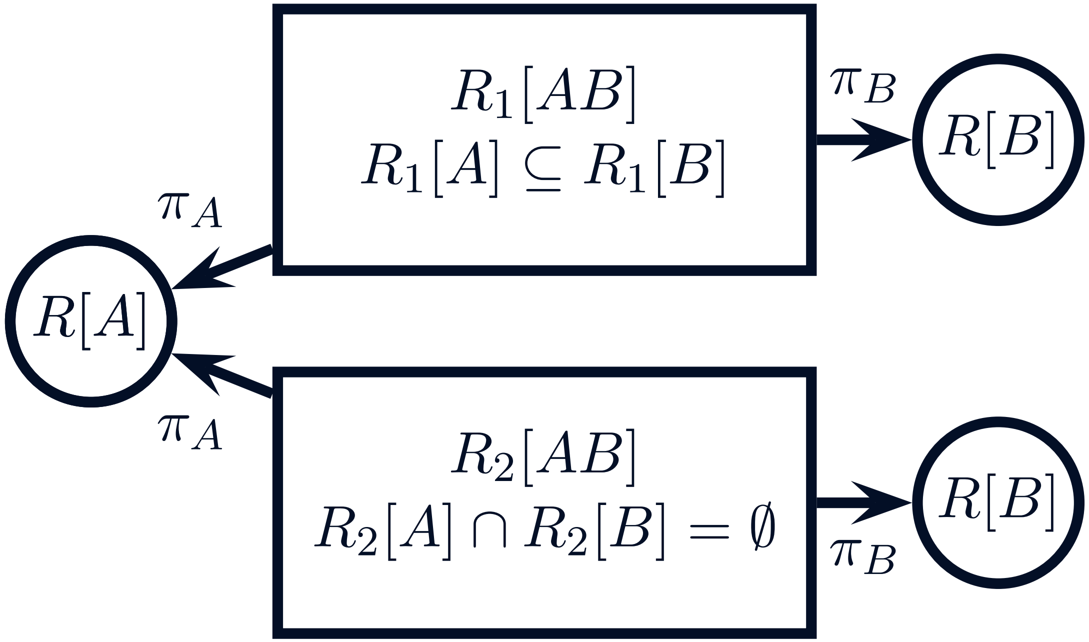
- Assume that all databases are finite.

Cyclicity Issues 2



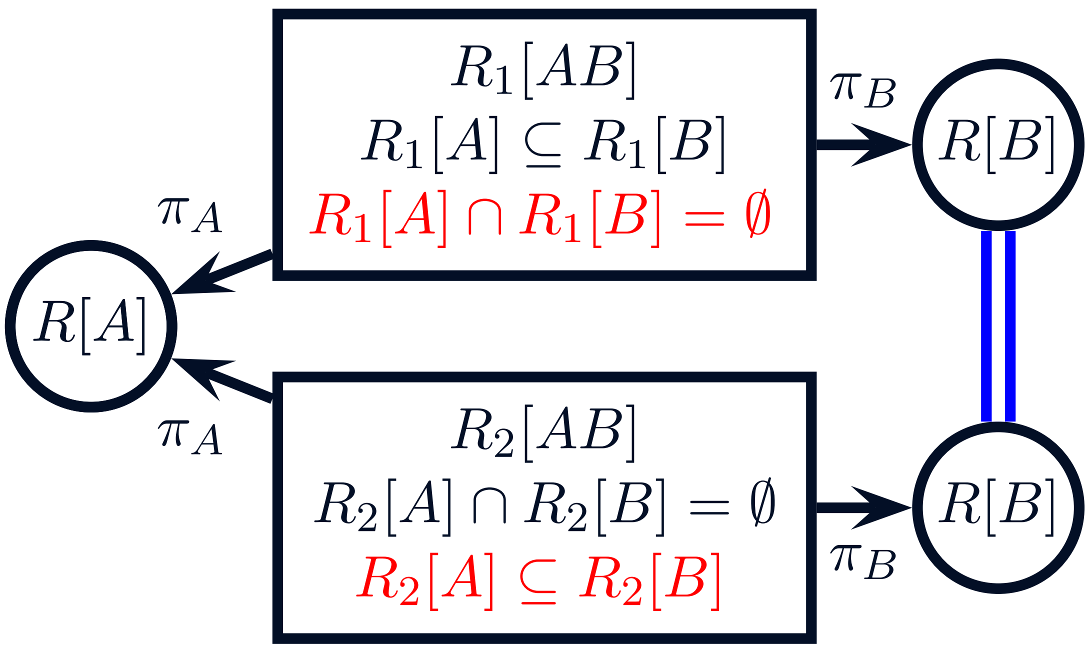
- Assume that all databases are finite.
- Connecting one set of ports causes no problems.

Cyclicity Issues 2

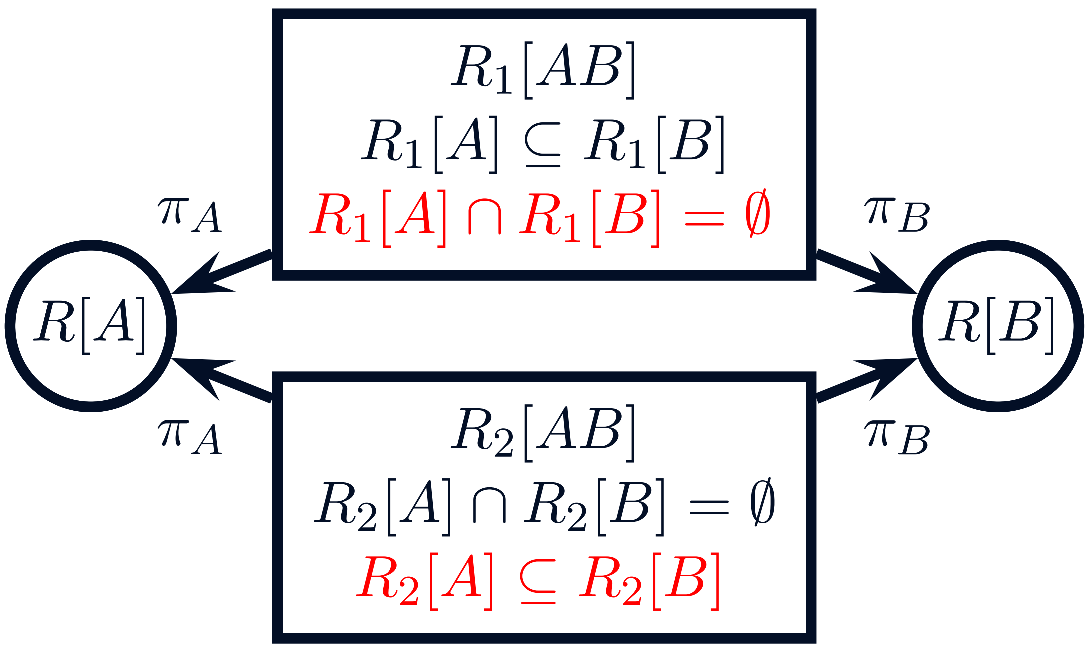


- Assume that all databases are finite.
- Connecting one set of ports causes no problems.

Cyclicity Issues 2

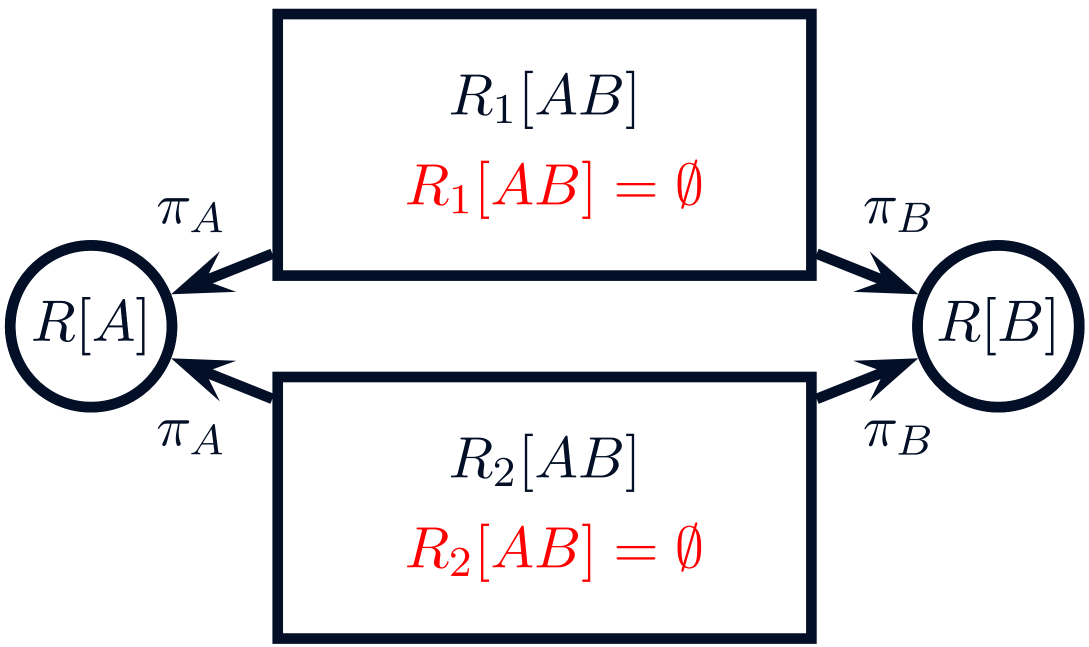


- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

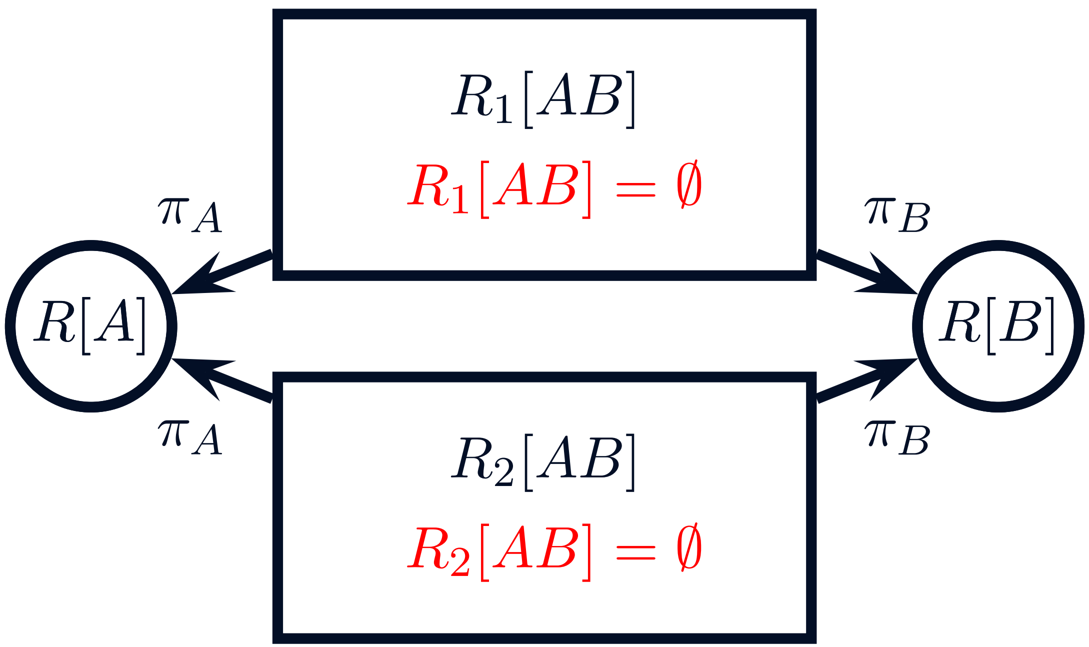
Cyclicity Issues 2



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

- The only legal component states are the empty relations.

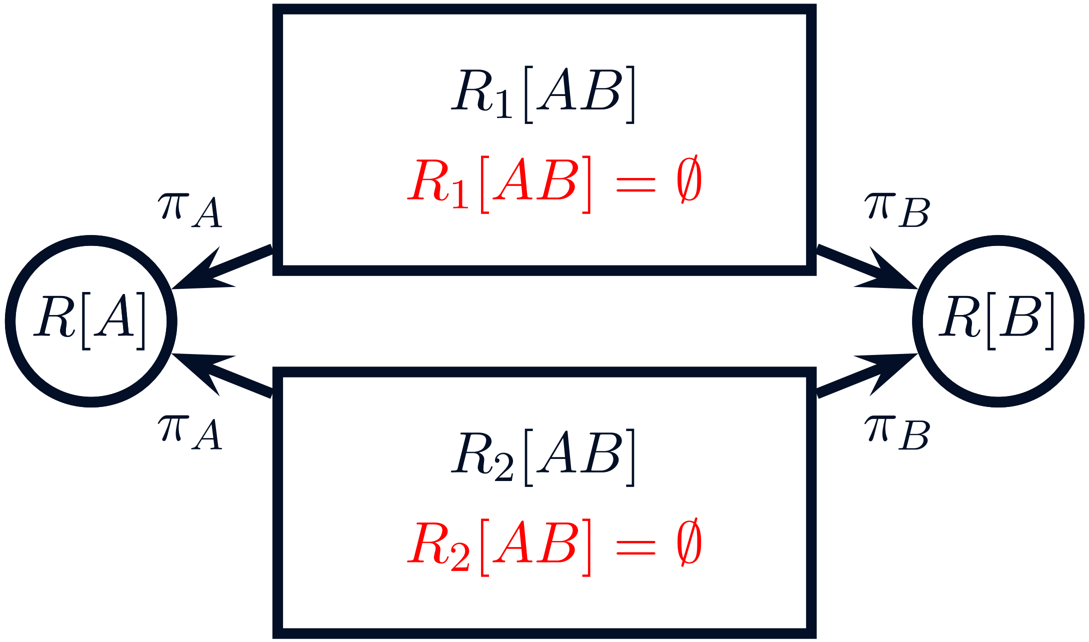
Cyclicity Issues 2



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

- The only legal component states are the empty relations.
- The existing ports are no longer views!

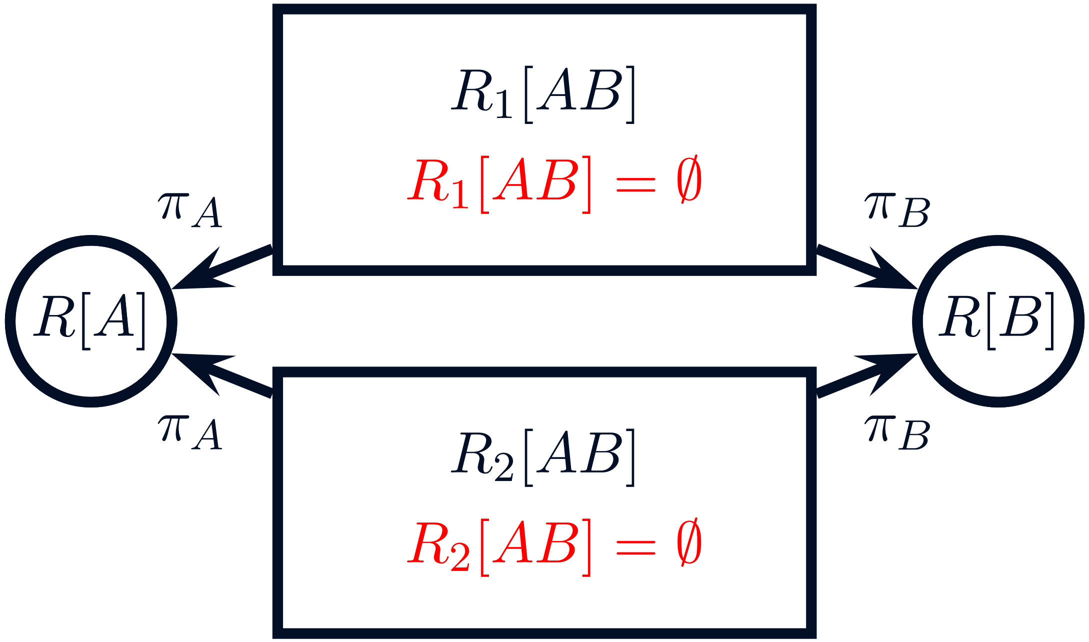
Cyclicity Issues 2



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

- The only legal component states are the empty relations.
- The existing ports are no longer views!
- To maintain surjectivity of the view mappings, the definitions of the port views must be changed.

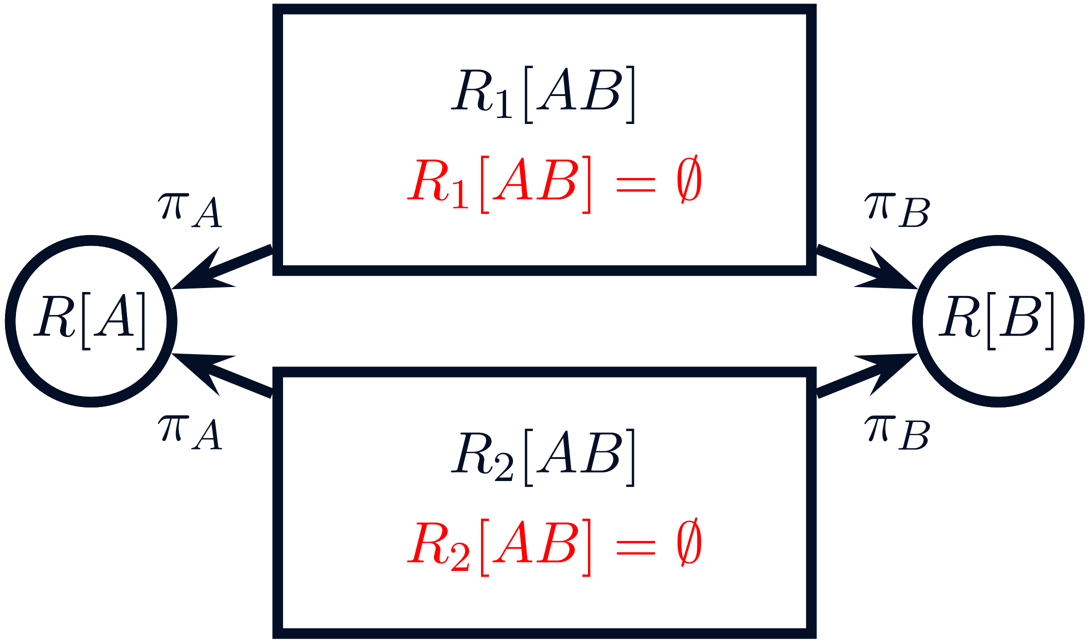
Cyclicity Issues 2



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

- The only legal component states are the empty relations.
- The existing ports are no longer views!
- To maintain surjectivity of the view mappings, the definitions of the port views must be changed.
- This type of cyclic interconnection is very difficult to manage.

Cyclicity Issues 2



- Assume that all databases are finite.
- Connecting one set of ports causes no problems.
- Connecting a second set introduces new dependencies on the component schemata.

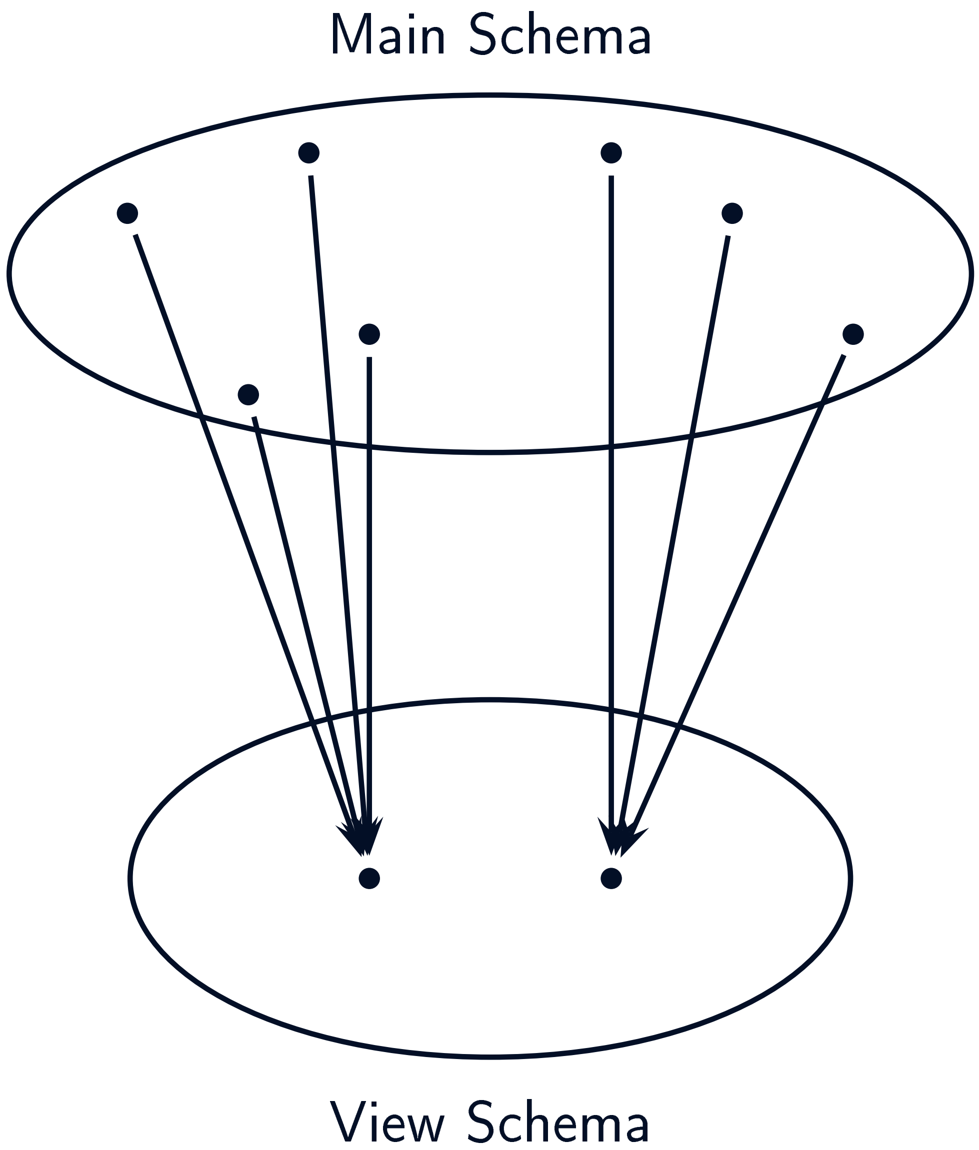
- The only legal component states are the empty relations.
- The existing ports are no longer views!
- To maintain surjectivity of the view mappings, the definitions of the port views must be changed.
- This type of cyclic interconnection is very difficult to manage.

$\Rightarrow$ Cyclicity is a major issue to be addressed in the theory.

Basic Update Support via Cooperation

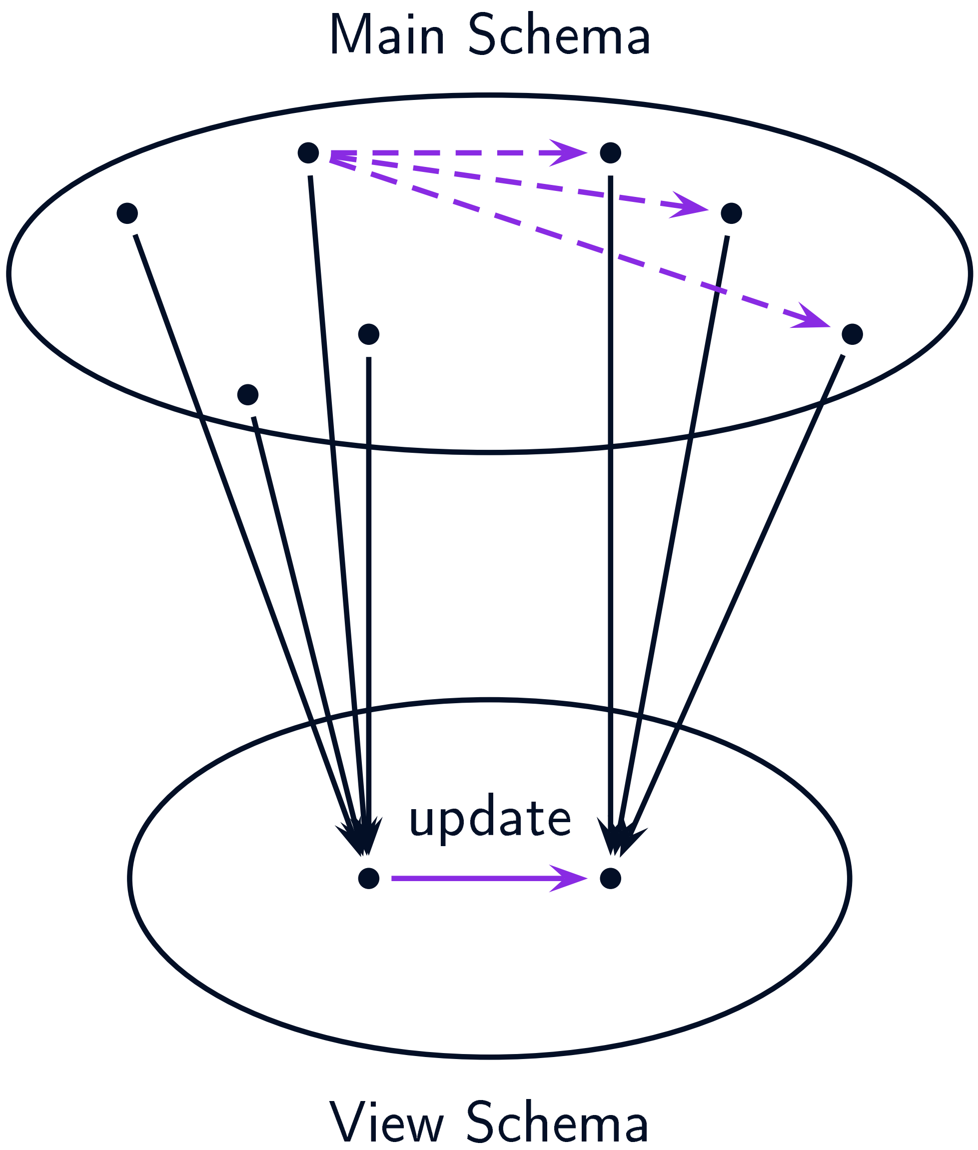
The Update Problem for Database Views

- On the underlying states, the view mapping is generally *surjective* (onto) but not *injective* (one-to-one).



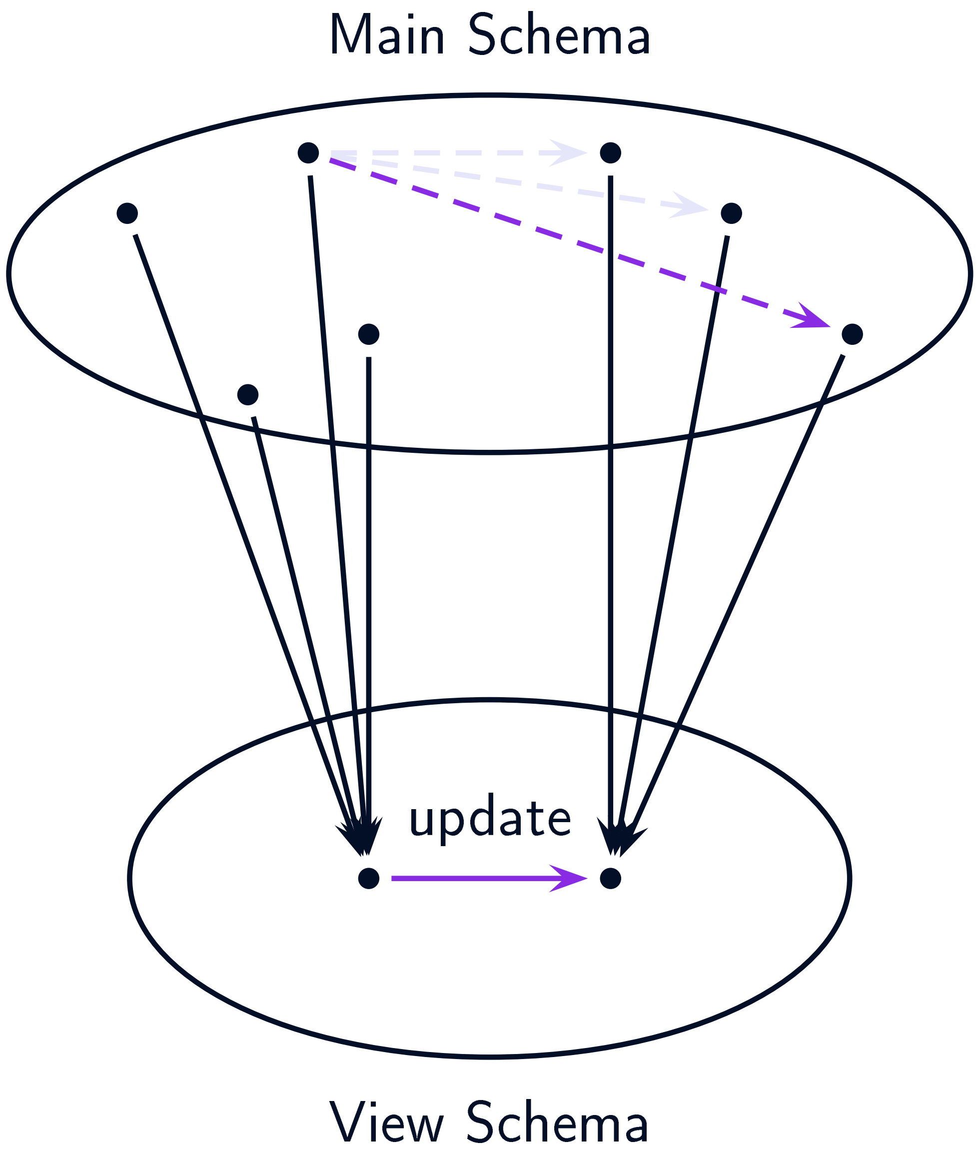
The Update Problem for Database Views

- On the underlying states, the view mapping is generally *surjective* (onto) but not *injective* (one-to-one).
- Thus, a view update has many possible *reflections* to the main schema.

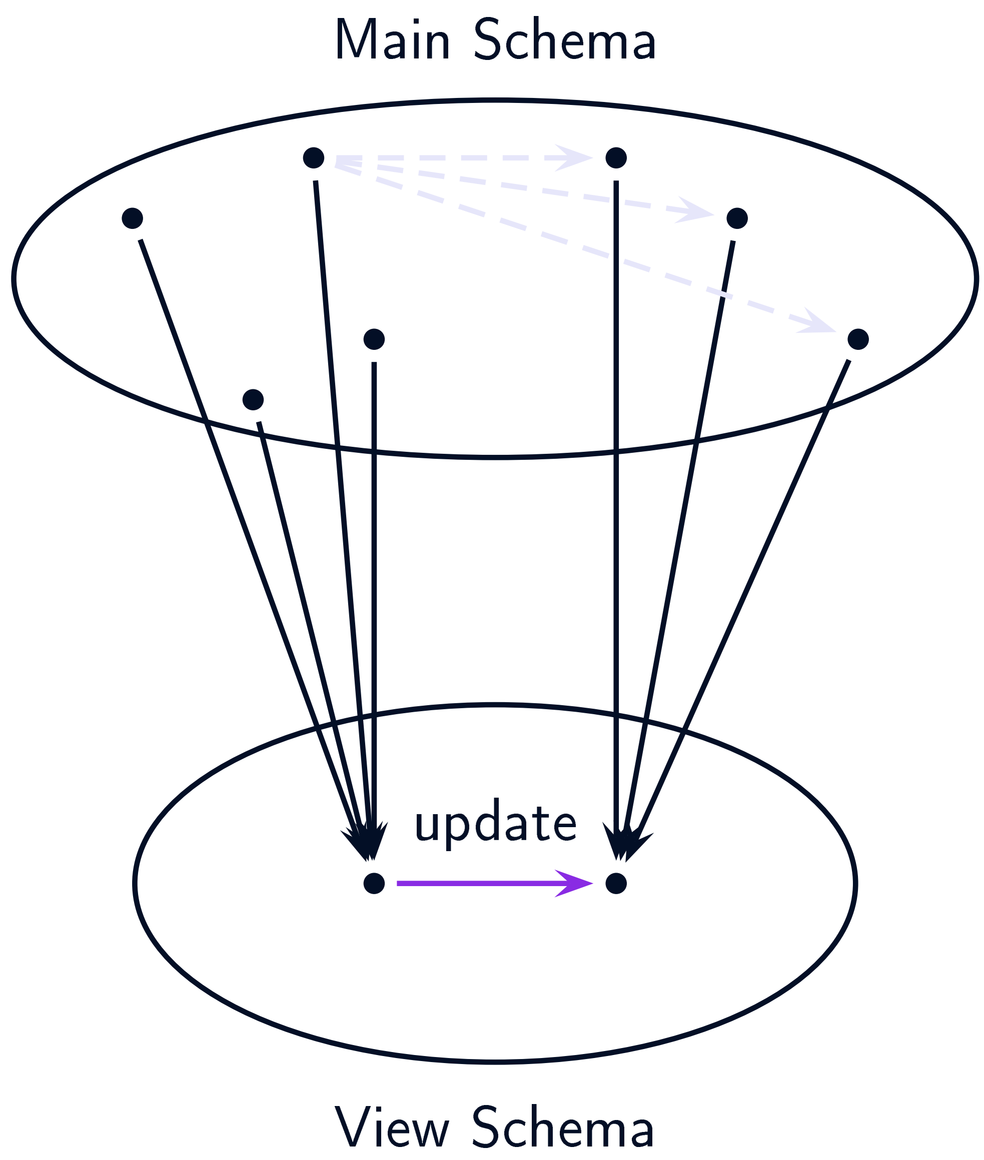


The Update Problem for Database Views

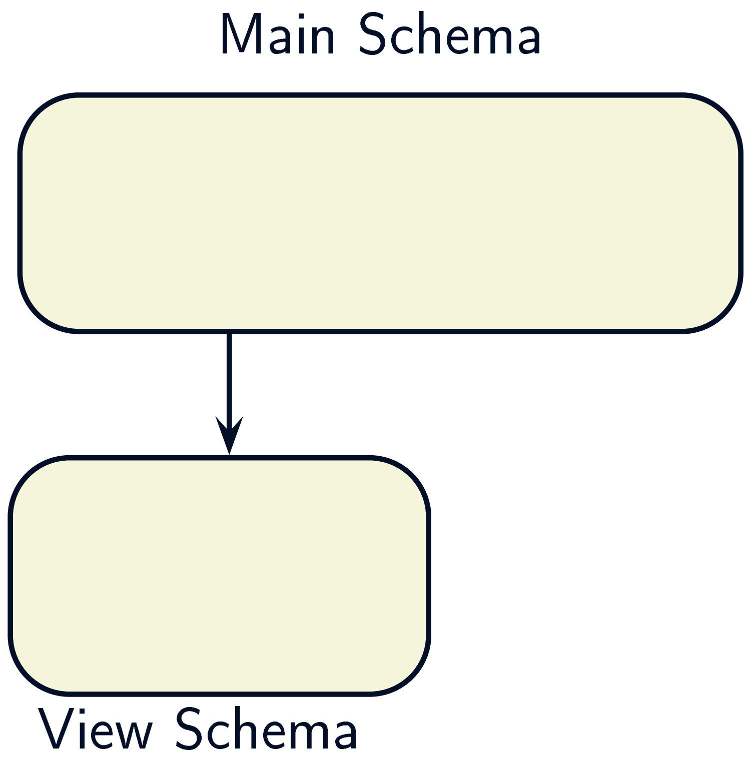
- On the underlying states, the view mapping is generally *surjective* (onto) but not *injective* (one-to-one).
- Thus, a view update has many possible *reflections* to the main schema.
- The problem of identifying a suitable reflection is known as the *update translation problem* or *update reflection problem*.



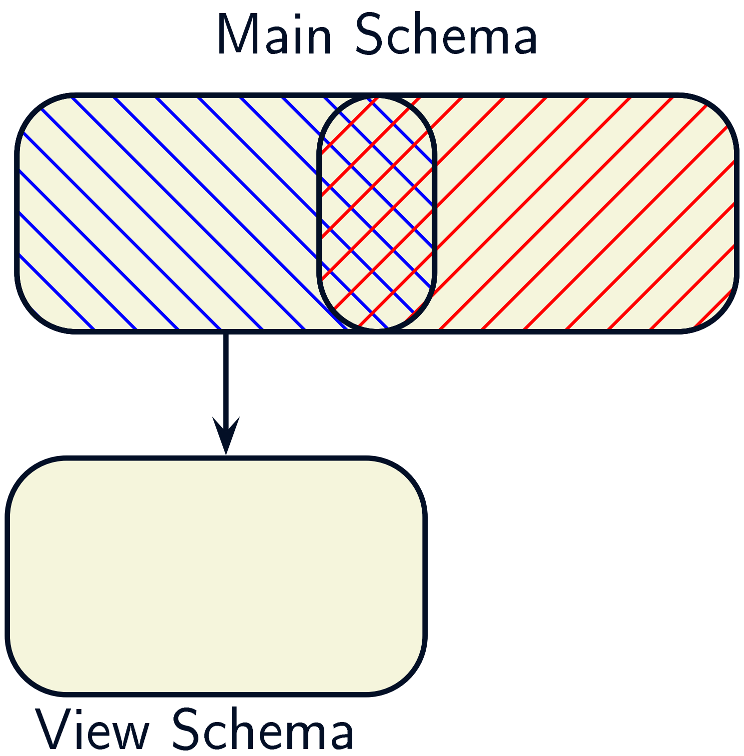
- On the underlying states, the view mapping is generally *surjective* (onto) but not *injective* (one-to-one).
- Thus, a view update has many possible *reflections* to the main schema.
- The problem of identifying a suitable reflection is known as the *update translation problem* or *update reflection problem*.
- With a reasonable definition of suitability, it may not be the case that every view update has a suitable translation.



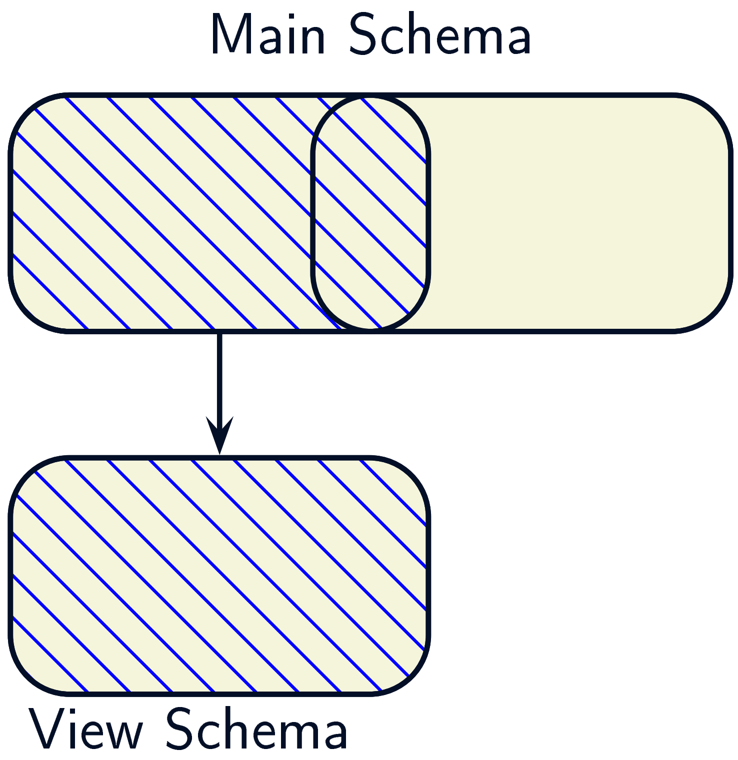
The Constant-Complement Strategy



- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.

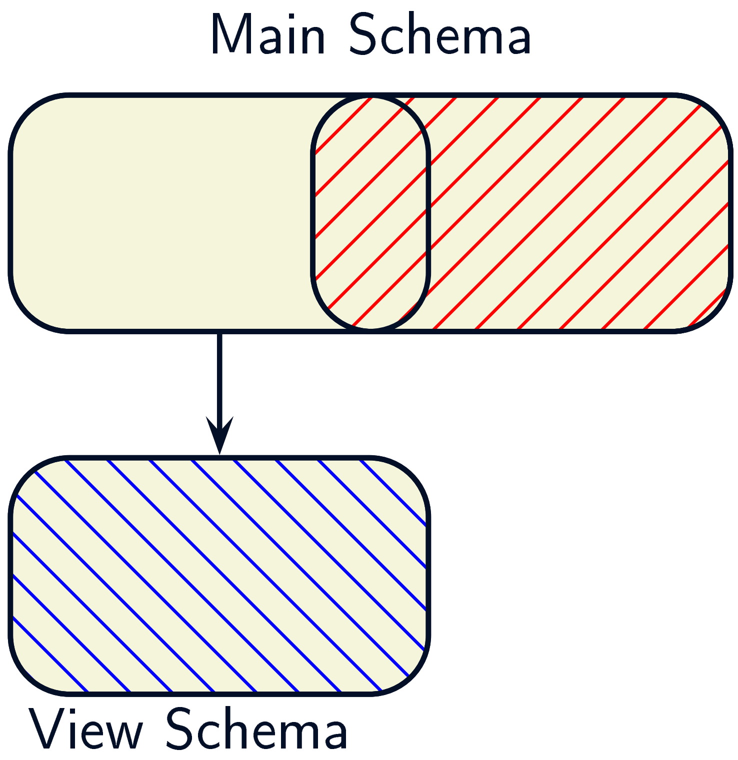


- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.
- One is isomorphic to the view schema and tracks its updates exactly.

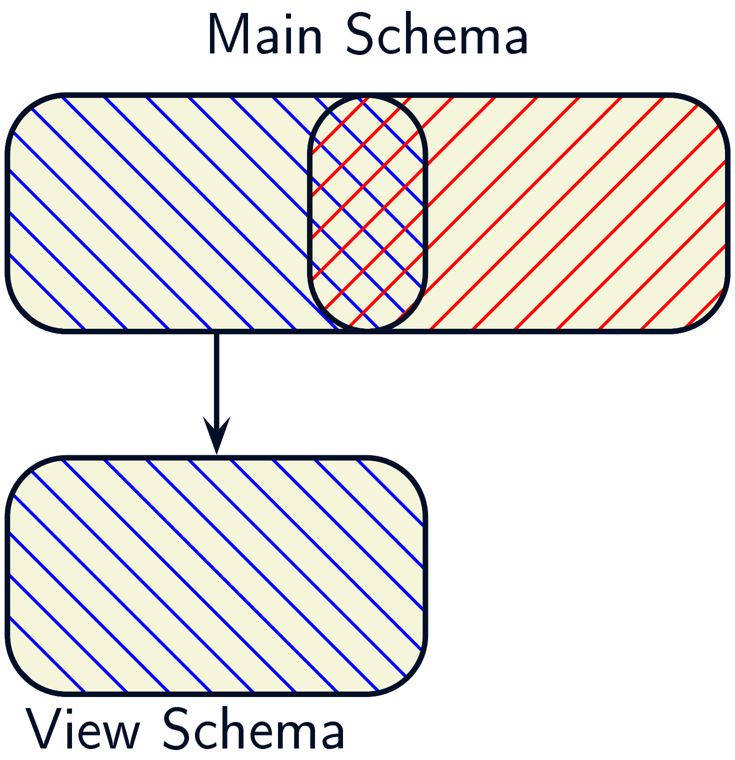


The Constant-Complement Strategy

- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.
- One is isomorphic to the view schema and tracks its updates exactly.
- The other is held constant for all updates to the view.

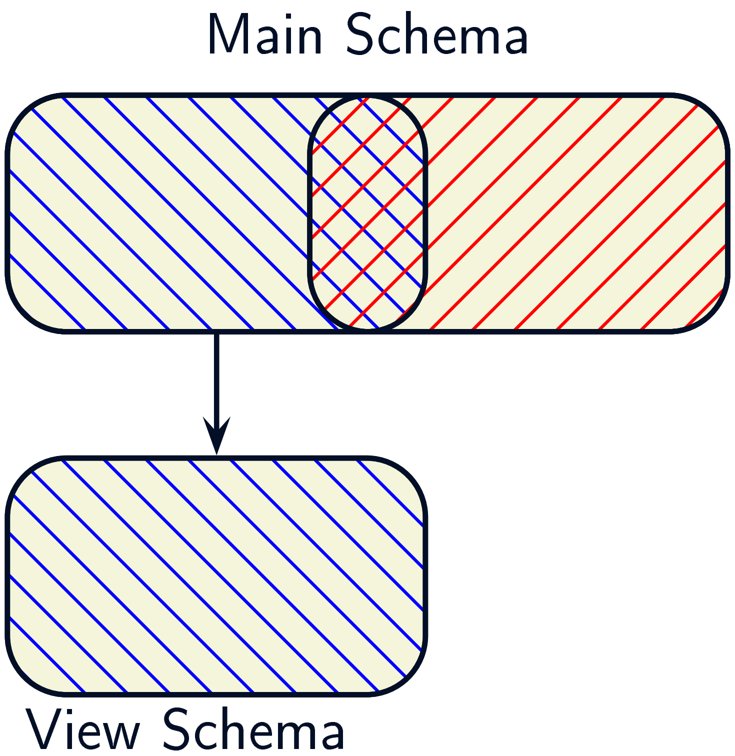


- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.
- One is isomorphic to the view schema and tracks its updates exactly.
- The other is held constant for all updates to the view.
- It can be shown [Hegner 03] that this strategy is precisely that which avoids all *update anomalies*.



The Constant-Complement Strategy

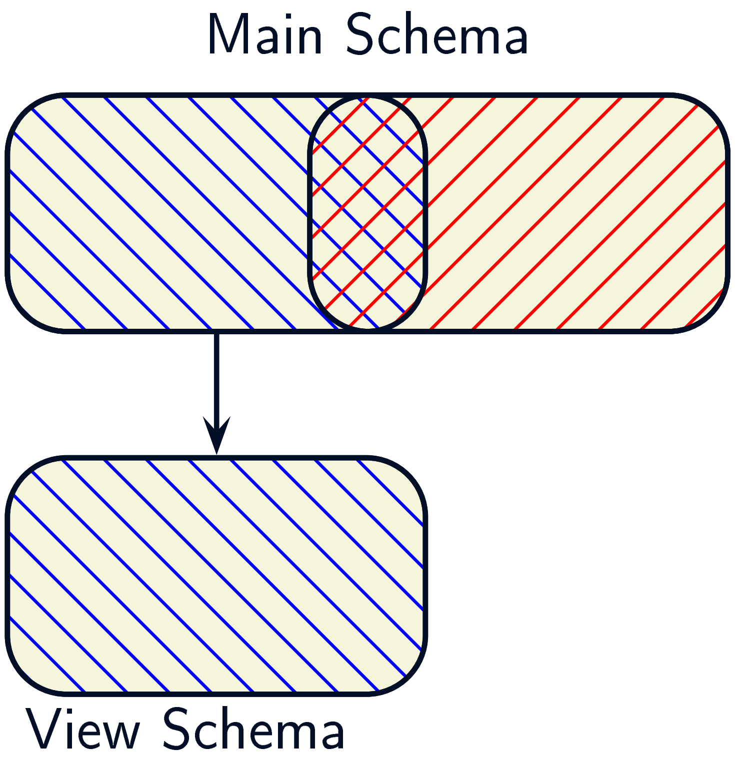
- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.
- One is isomorphic to the view schema and tracks its updates exactly.
- The other is held constant for all updates to the view.
- It can be shown [Hegner 03] that this strategy is precisely that which avoids all *update anomalies*.



- Consequently, it is quite limited in the view updates which it allows.

The Constant-Complement Strategy

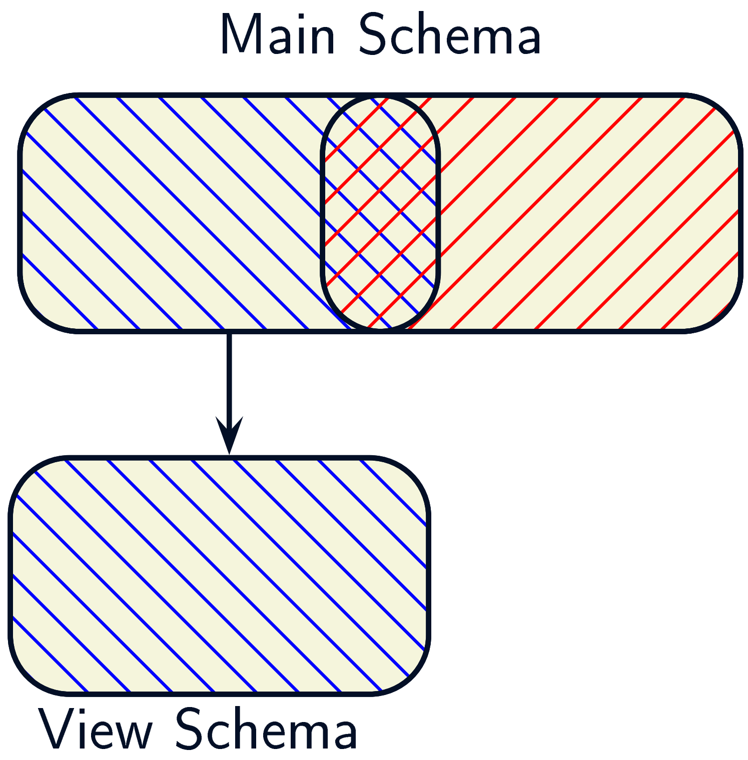
- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.
- One is isomorphic to the view schema and tracks its updates exactly.
- The other is held constant for all updates to the view.
- It can be shown [Hegner 03] that this strategy is precisely that which avoids all *update anomalies*.



- Consequently, it is quite limited in the view updates which it allows.

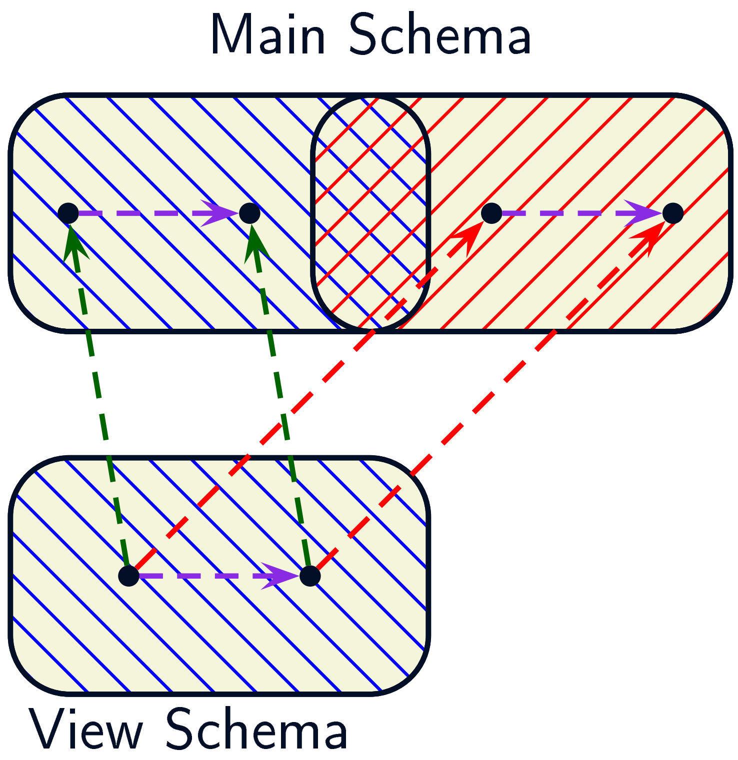
Question: How can updates which are not supported by constant complement be realized?

- Over the years, many extensions to the constant-complement strategy have been proposed; all share the following problems.



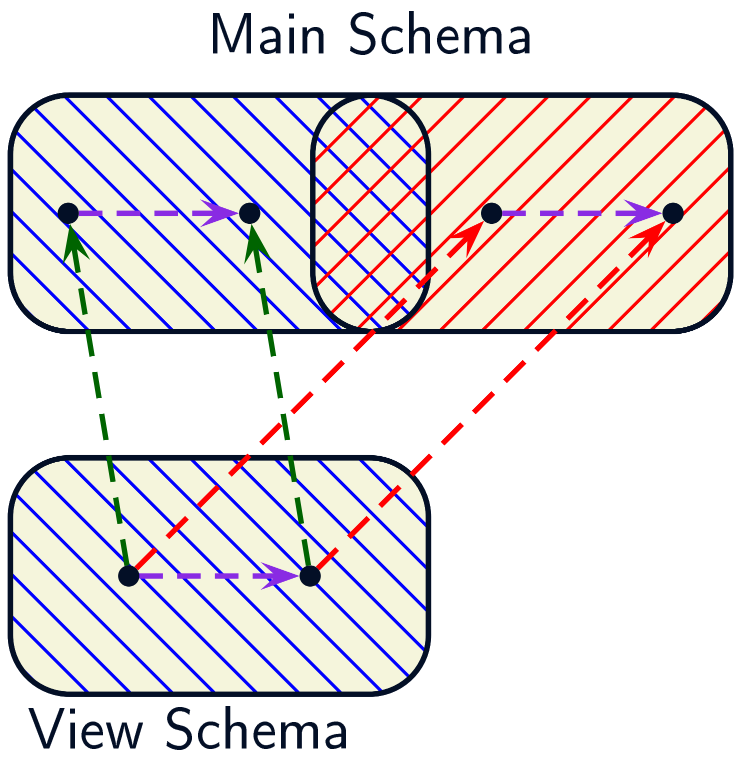
Moving Beyond the Constant-Complement Strategy

- Over the years, many extensions to the constant-complement strategy have been proposed; all share the following problems.
- Visibility problem: Part of the reflected update is not visible within the view.

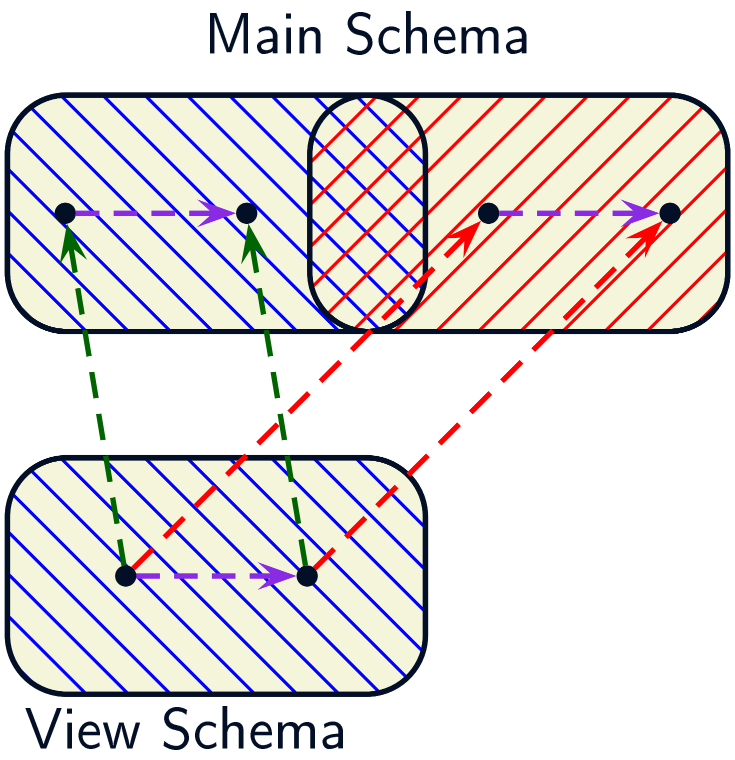


Moving Beyond the Constant-Complement Strategy

- Over the years, many extensions to the constant-complement strategy have been proposed; all share the following problems.
- Visibility problem: Part of the reflected update is not visible within the view.
- Permission problem: The user of the view lacks the necessary access privileges to effect the reflected update to the main schema.



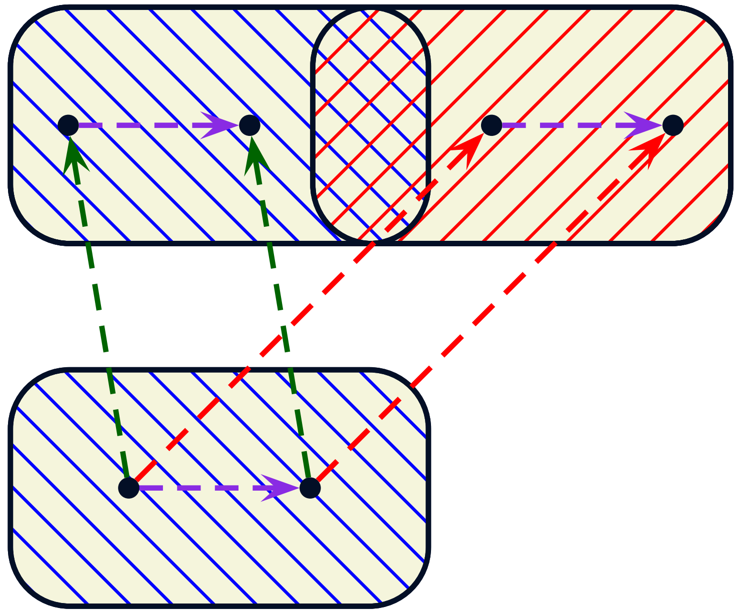
- Over the years, many extensions to the constant-complement strategy have been proposed; all share the following problems.
- Visibility problem: Part of the reflected update is not visible within the view.
- Permission problem: The user of the view lacks the necessary access privileges to effect the reflected update to the main schema.



Proposed Solution: *Update by cooperation*

Moving Beyond the Constant-Complement Strategy

- Over the years, many extensions to the constant-complement strategy have been proposed; all share the following problems.
- Visibility problem: Part of the reflected update is not visible within the view.
- Permission problem: The user of the view lacks the necessary access privileges to effect the reflected update to the main schema.

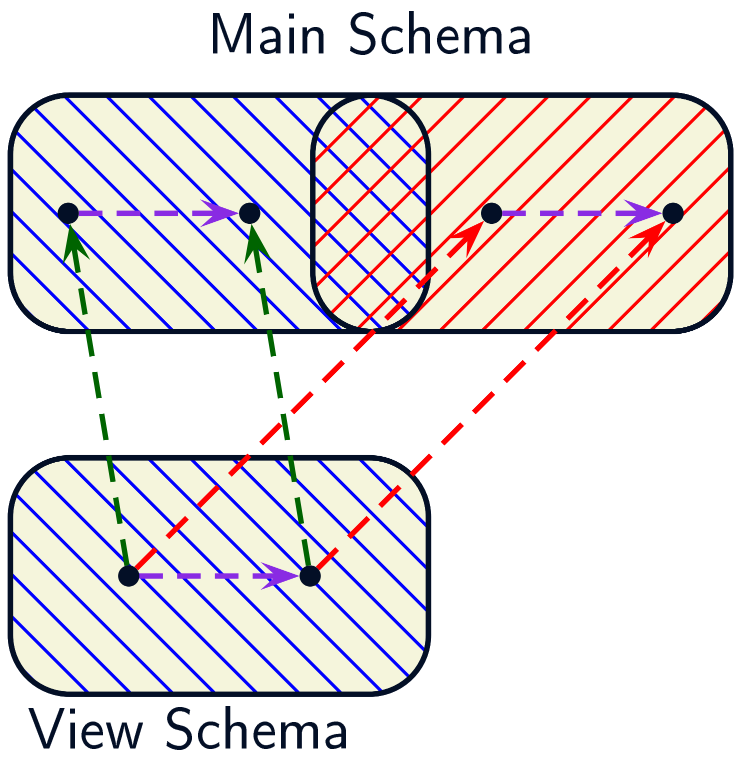


Proposed Solution: *Update by cooperation*

- The user of the view enlists the cooperation of other users to address both the visibility problem and the permission problem.

Moving Beyond the Constant-Complement Strategy

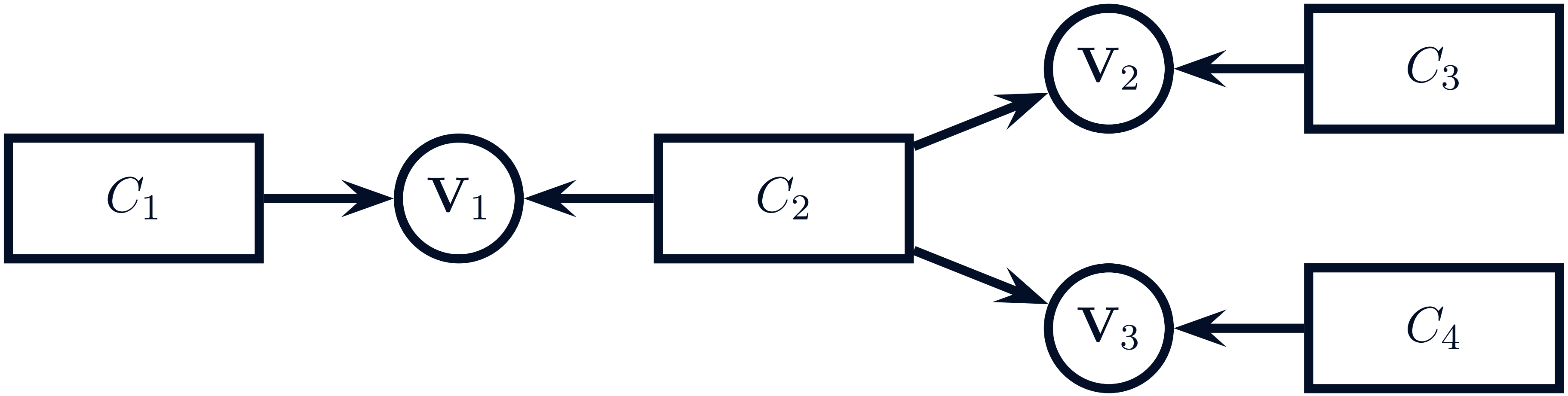
- Over the years, many extensions to the constant-complement strategy have been proposed; all share the following problems.
- Visibility problem: Part of the reflected update is not visible within the view.
- Permission problem: The user of the view lacks the necessary access privileges to effect the reflected update to the main schema.



Proposed Solution: *Update by cooperation*

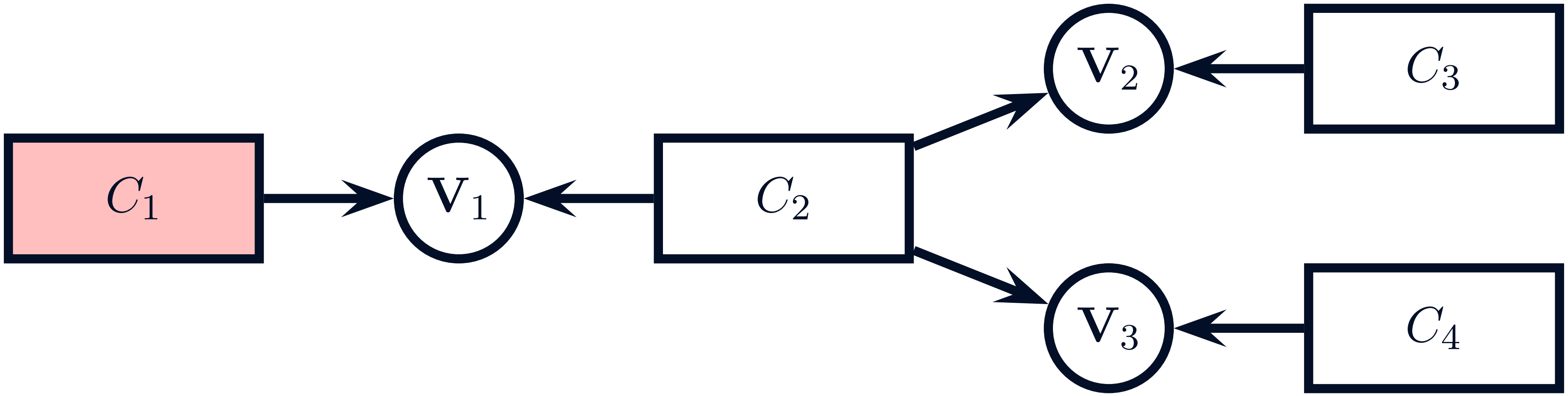
- The user of the view enlists the cooperation of other users to address both the visibility problem and the permission problem.
- All users operate within the limits of their vision of the main schema and their access rights.

The Propagation of Updates through Components



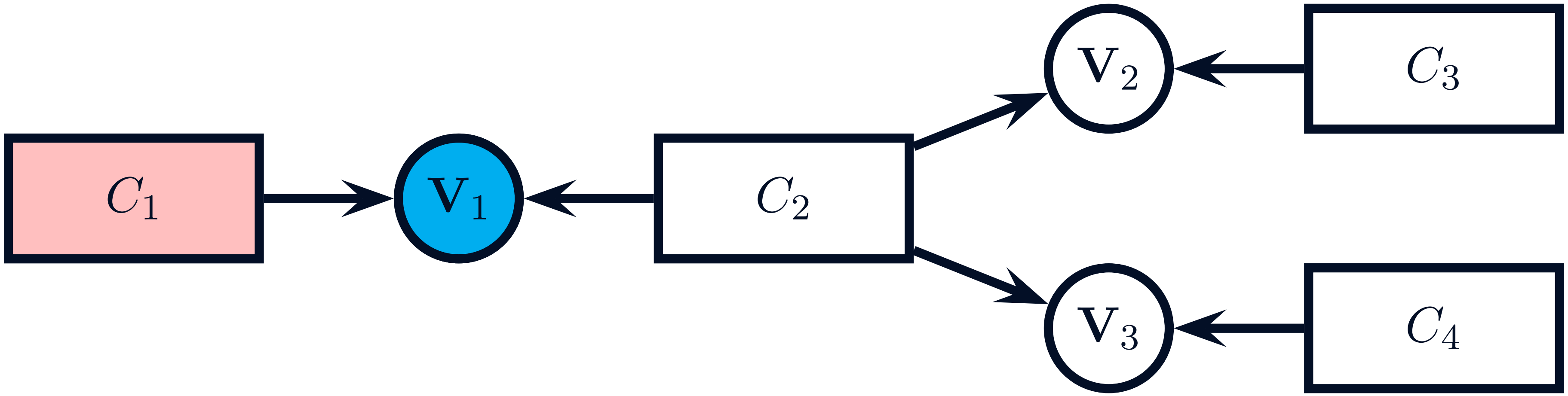
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.



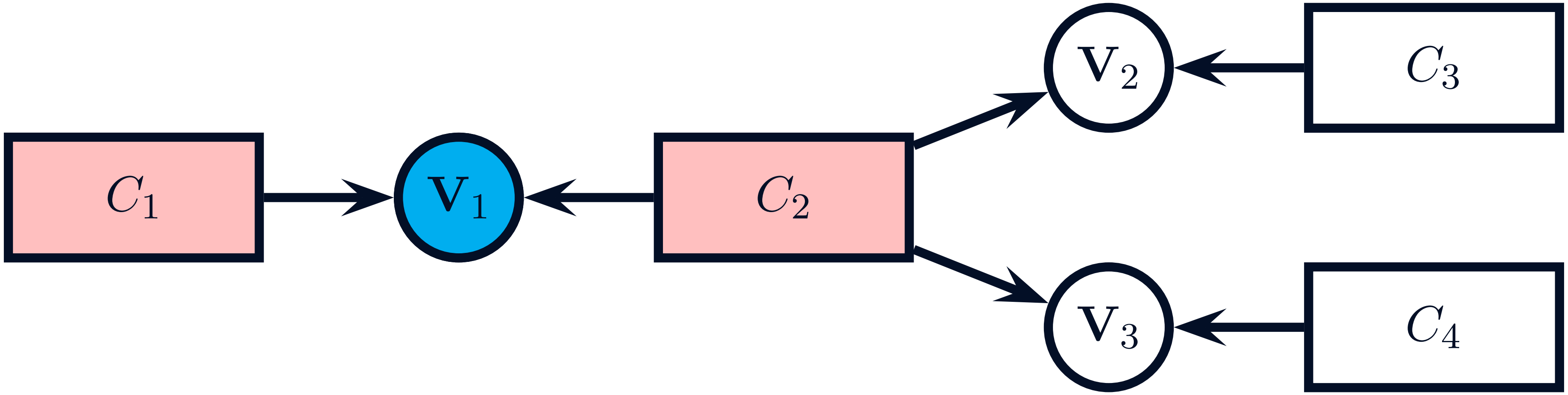
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.



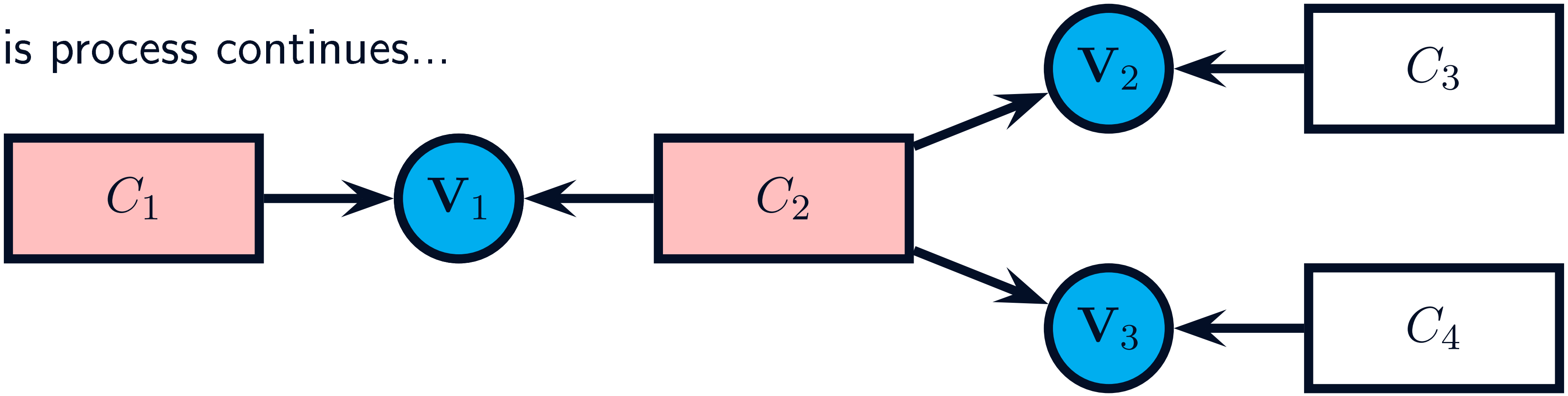
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.



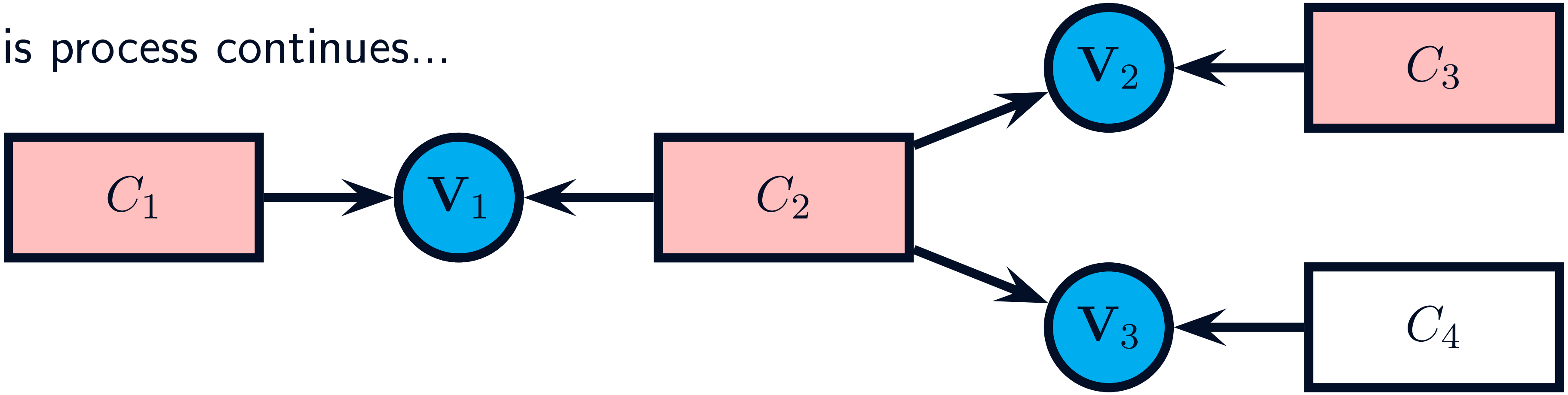
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



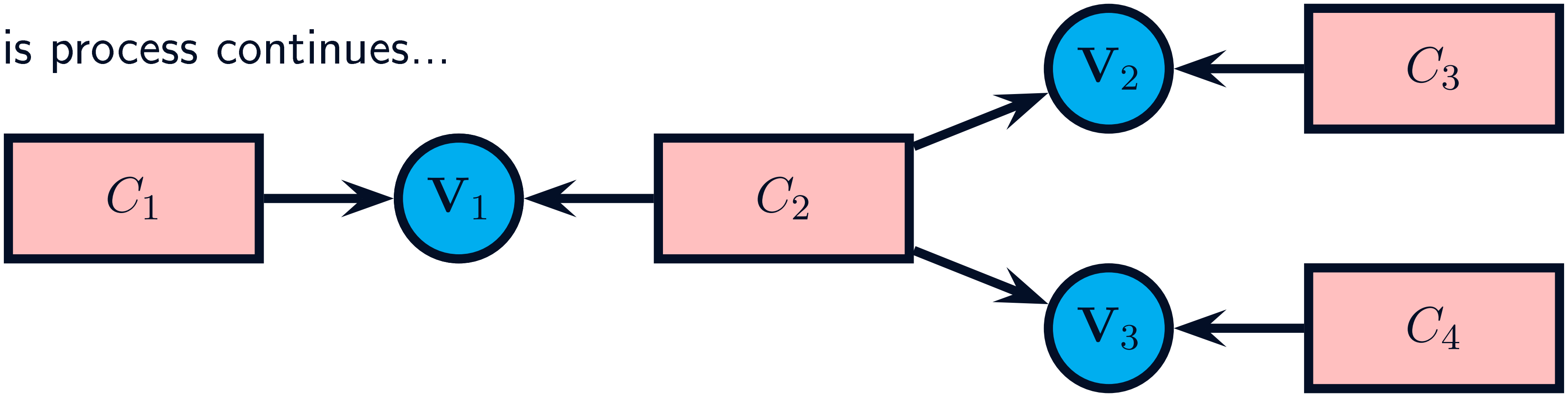
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



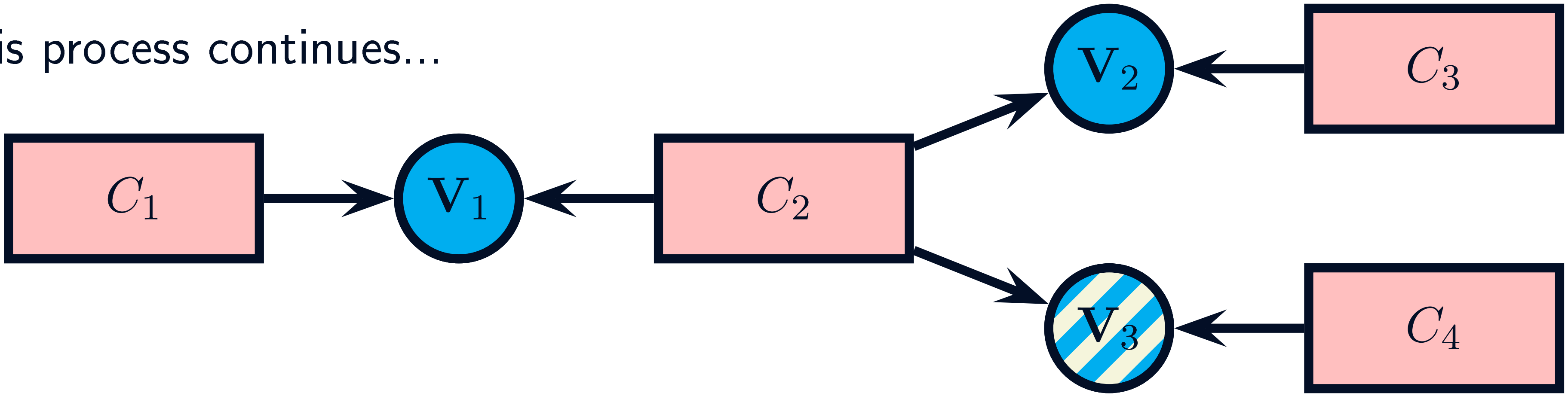
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



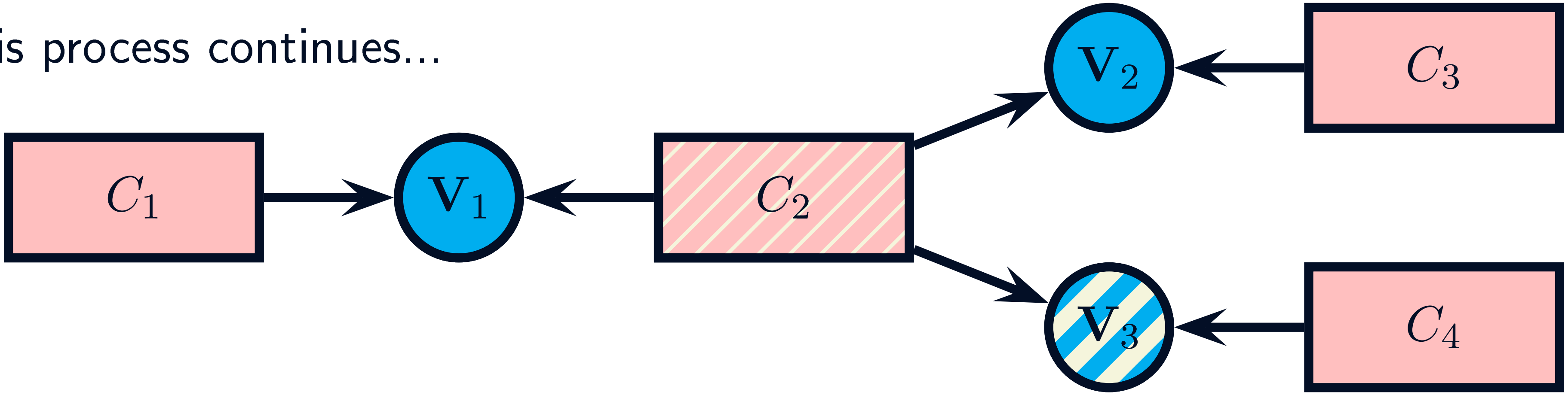
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...

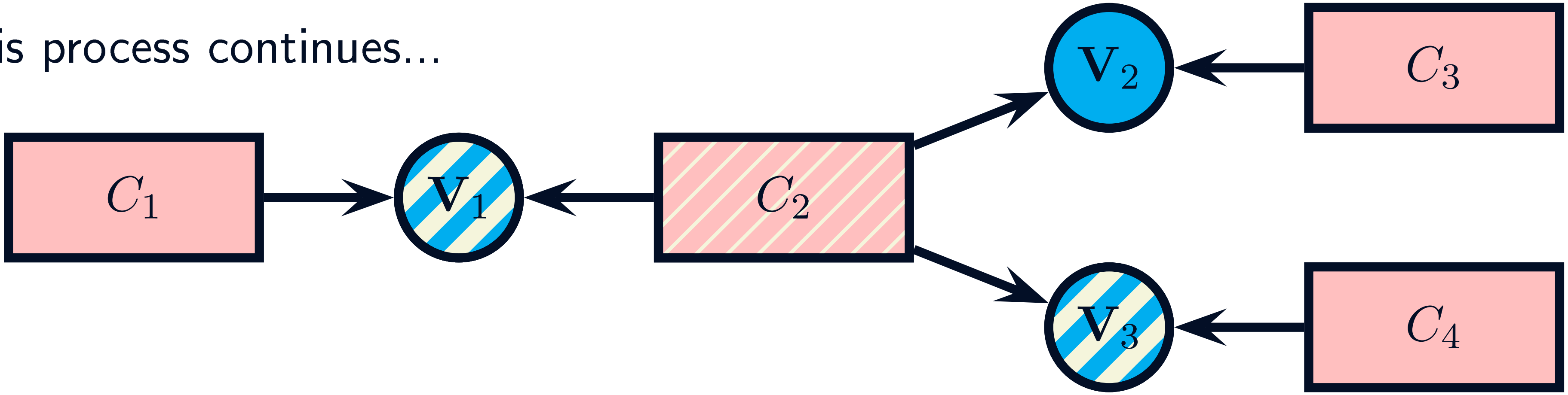


The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...

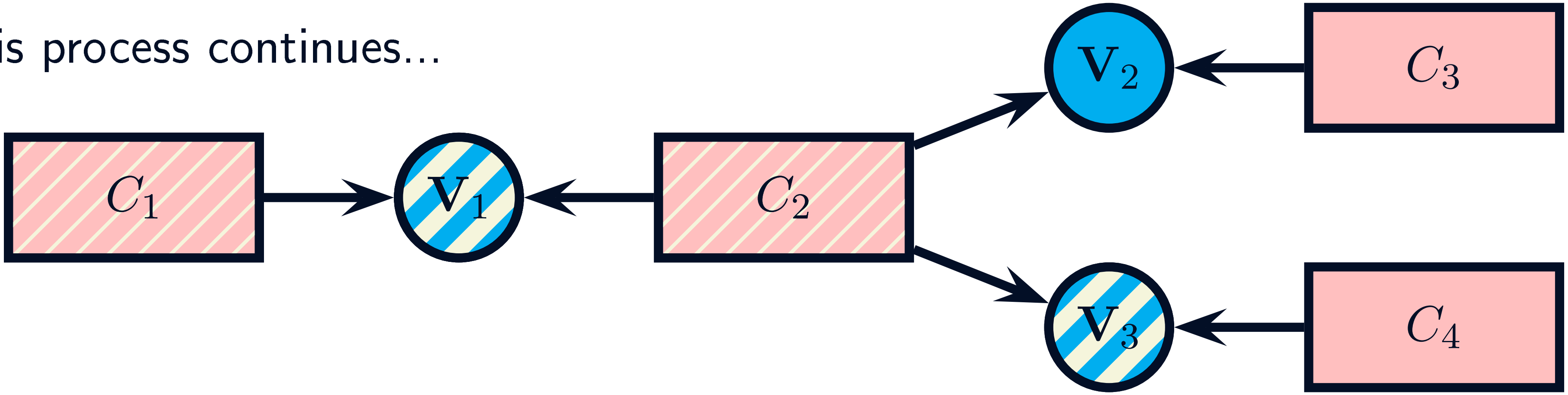


- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



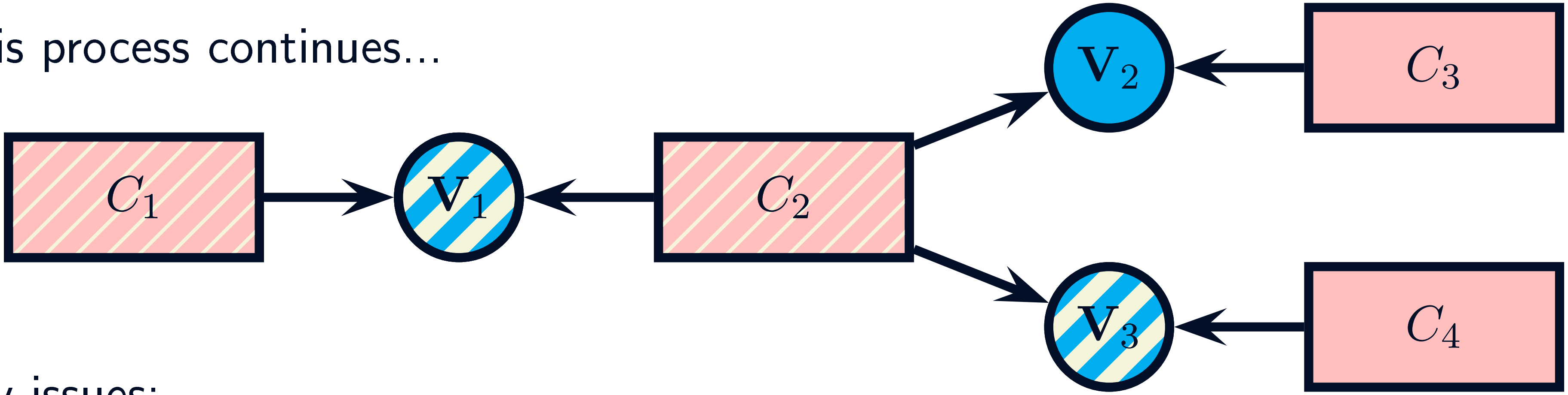
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



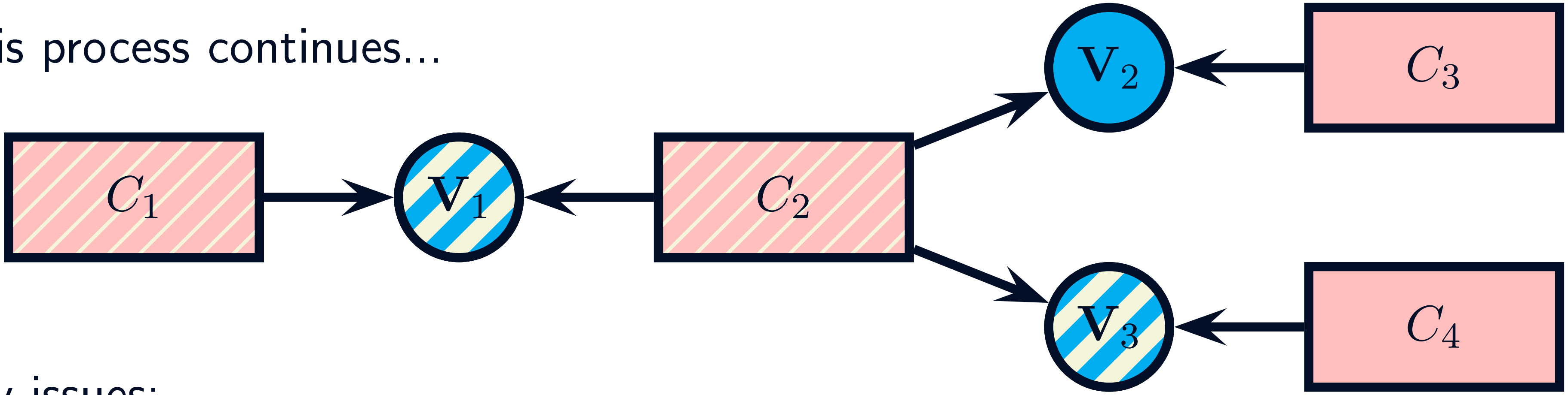
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



- Key issues:

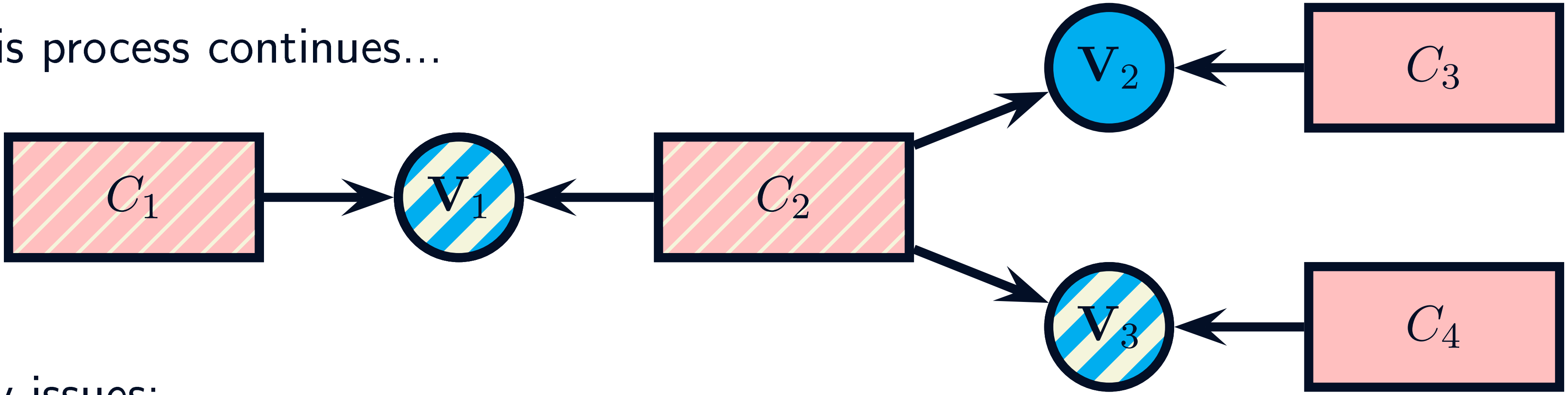
- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



- Key issues:
  - *Database consistency*: Actual database update must be deferred until the negotiation process is complete.

The Propagation of Updates through Components

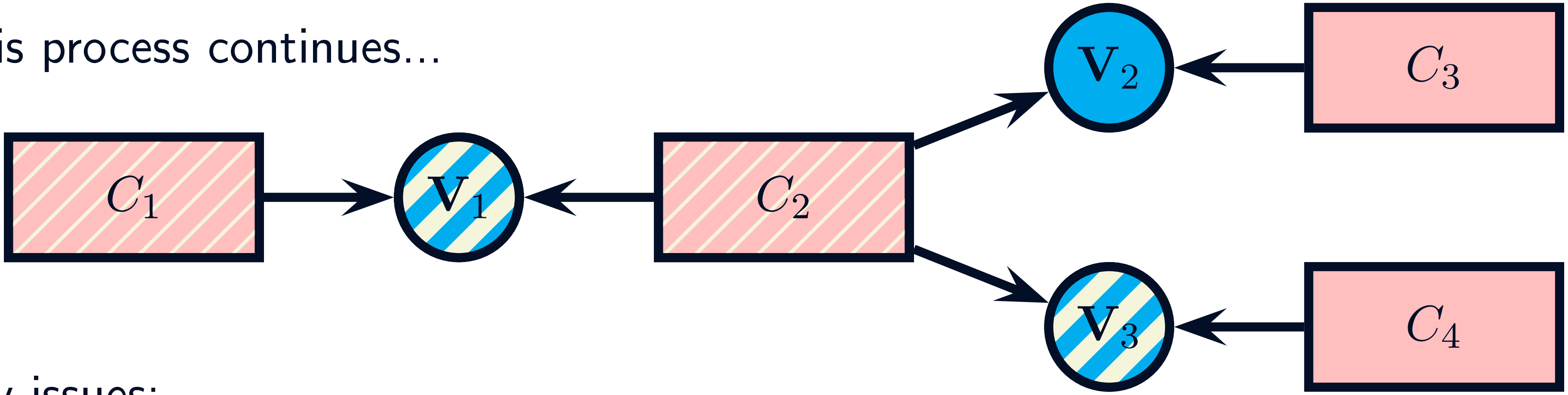
- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...



- Key issues:
  - *Database consistency*: Actual database update must be deferred until the negotiation process is complete.
  - *Termination*: The negotiation process must not go on endlessly.

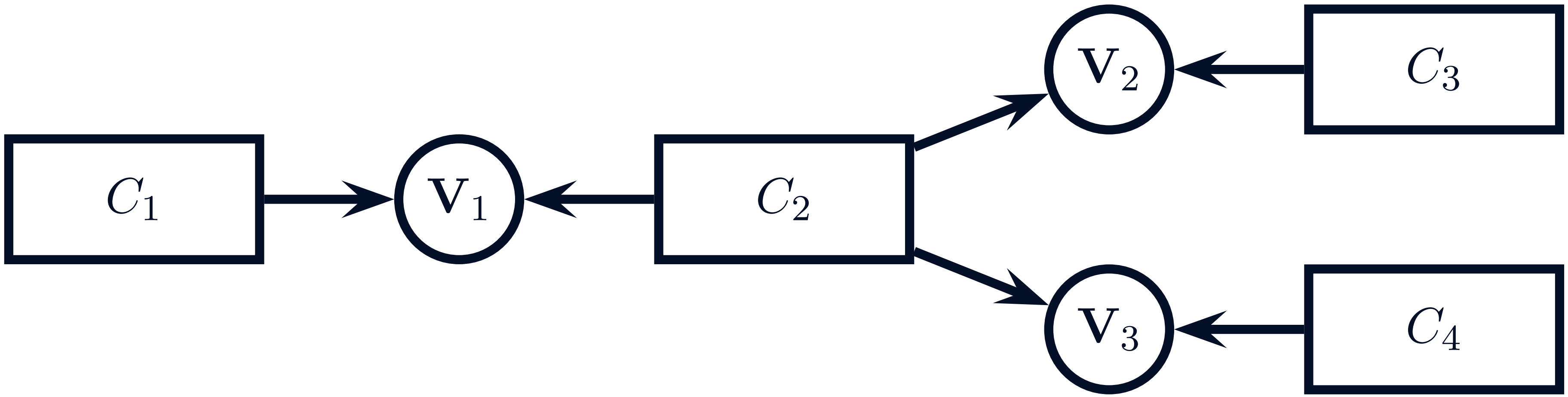
The Propagation of Updates through Components

- Suppose that an update to the schema of component C_1 is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to C_2.
- This process continues...

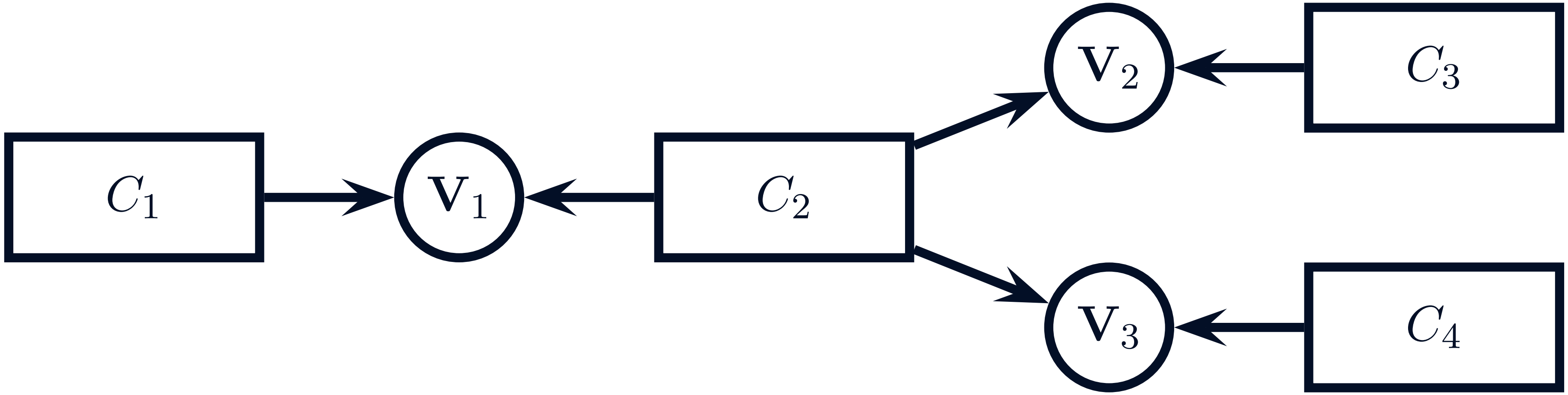


- Key issues:
  - *Database consistency*: Actual database update must be deferred until the negotiation process is complete.
  - *Termination*: The negotiation process must not go on endlessly.
- An architecture for the support of such *cooperative updates* is needed.

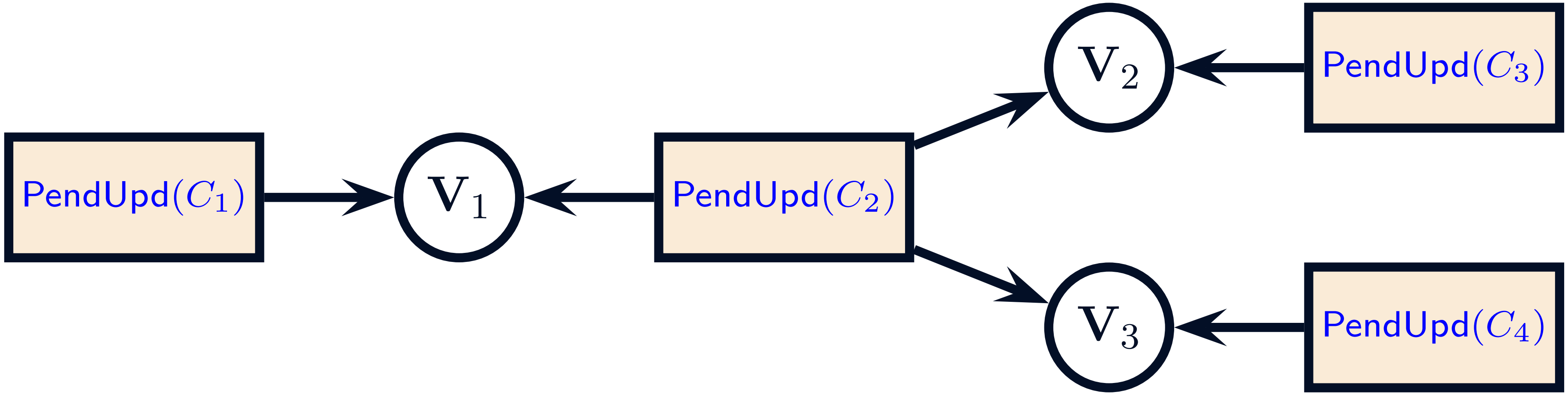
The Architecture of Cooperative Update Management



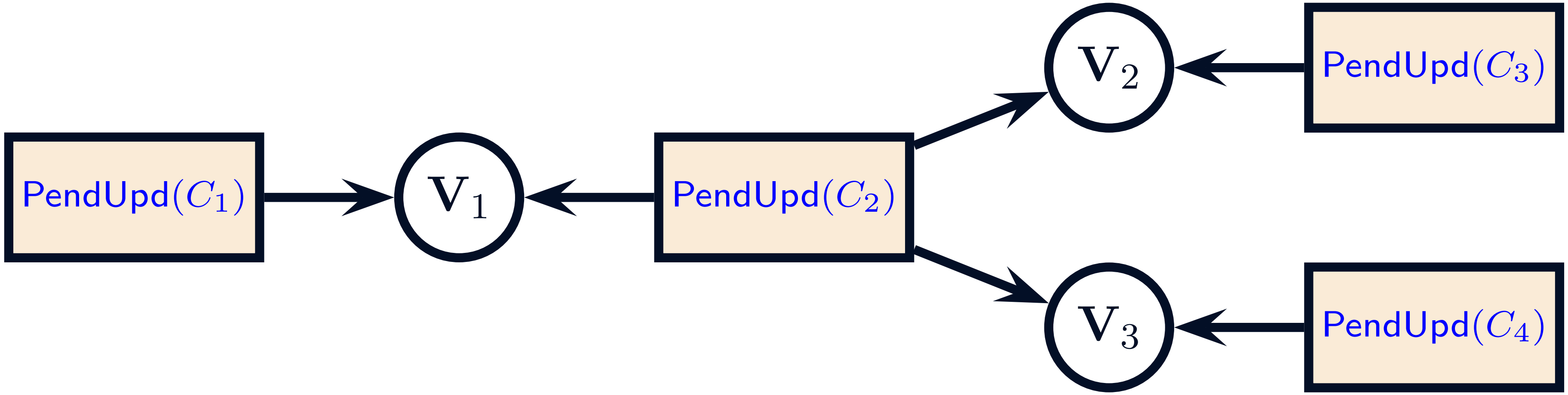
The Architecture of Cooperative Update Management



- To each component C_i corresponds a *pending-update register* $\mathsf{PendUpd}(C_i)$.

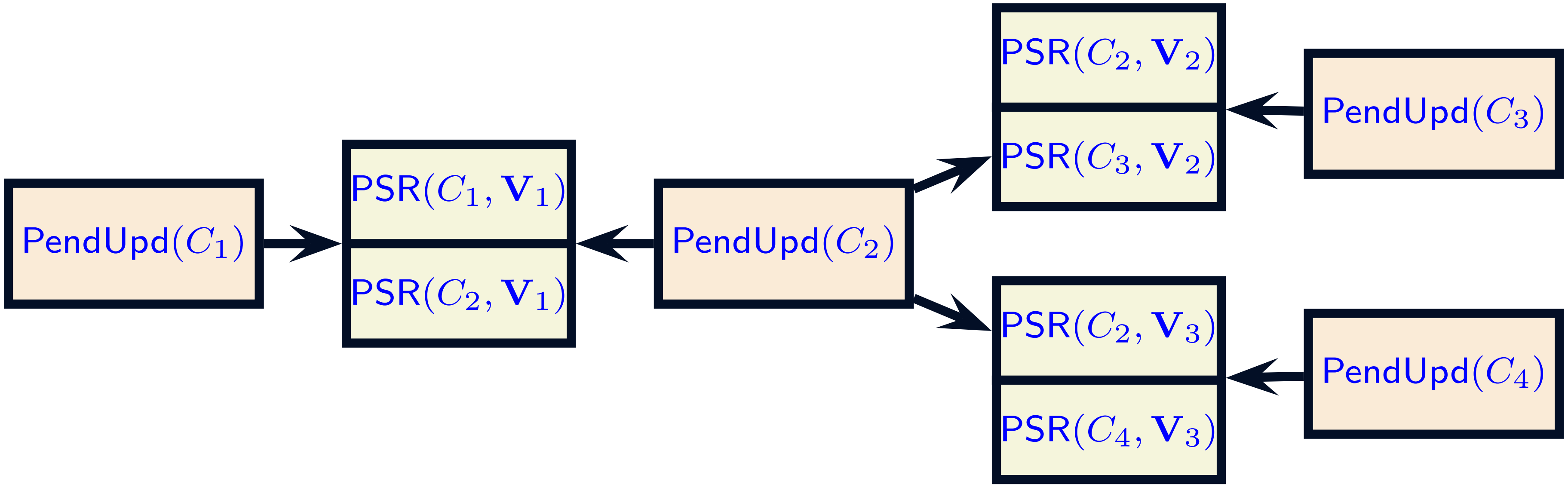


- To each component C_i corresponds a *pending-update register* PendUpd(C_i).



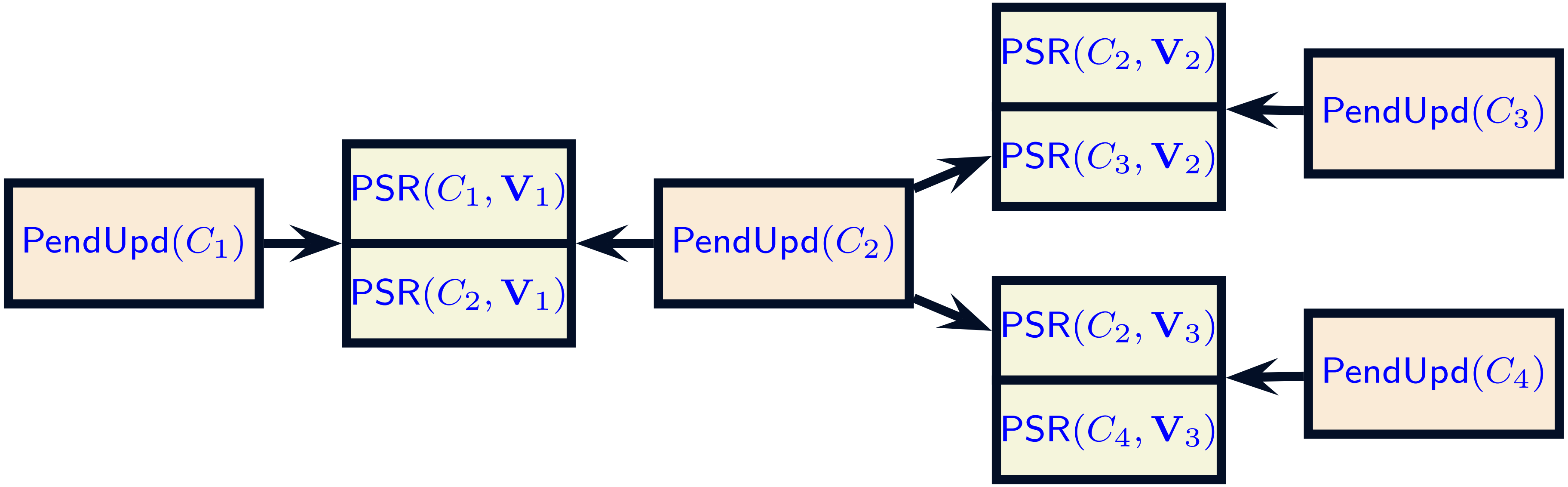
- To each component C_i corresponds a *pending-update register* PendUpd(C_i).
- To each view schema $\mathbf{V}_i$ is associated a *port-status register* PSR$(C_j, \mathbf{V}_i)$ for each component C_j which is connected to it.

The Architecture of Cooperative Update Management



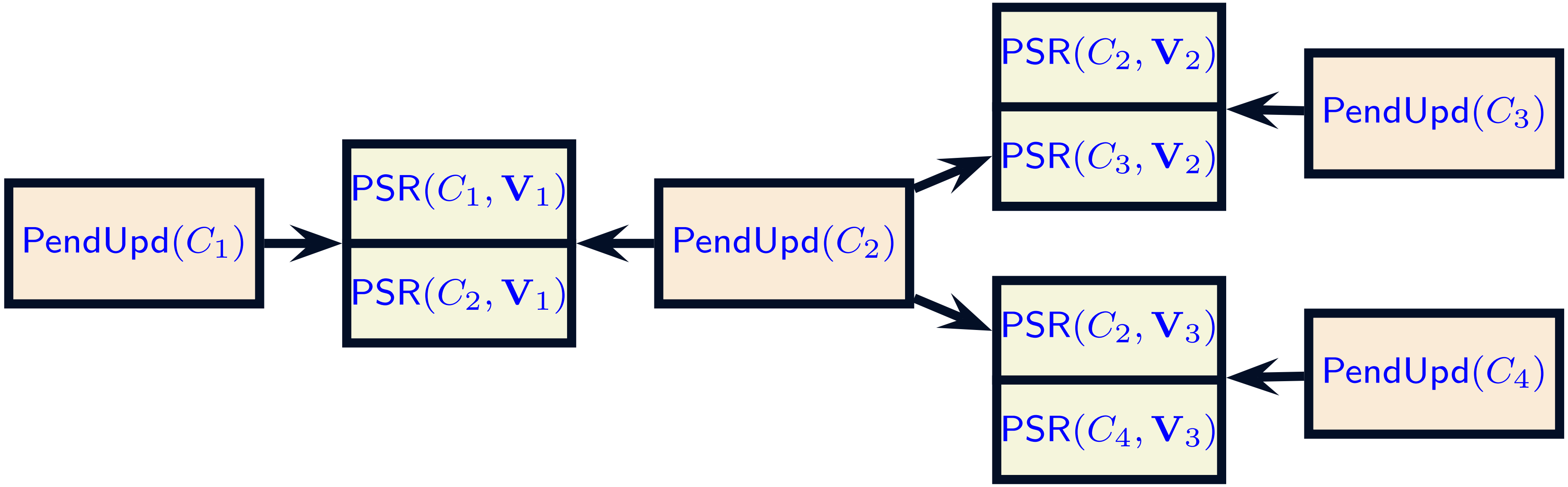
- To each component C_i corresponds a *pending-update register* PendUpd(C_i).
- To each view schema $\mathbf{V}_i$ is associated a *port-status register* PSR$(C_j, \mathbf{V}_i)$ for each component C_j which is connected to it.

The Architecture of Cooperative Update Management



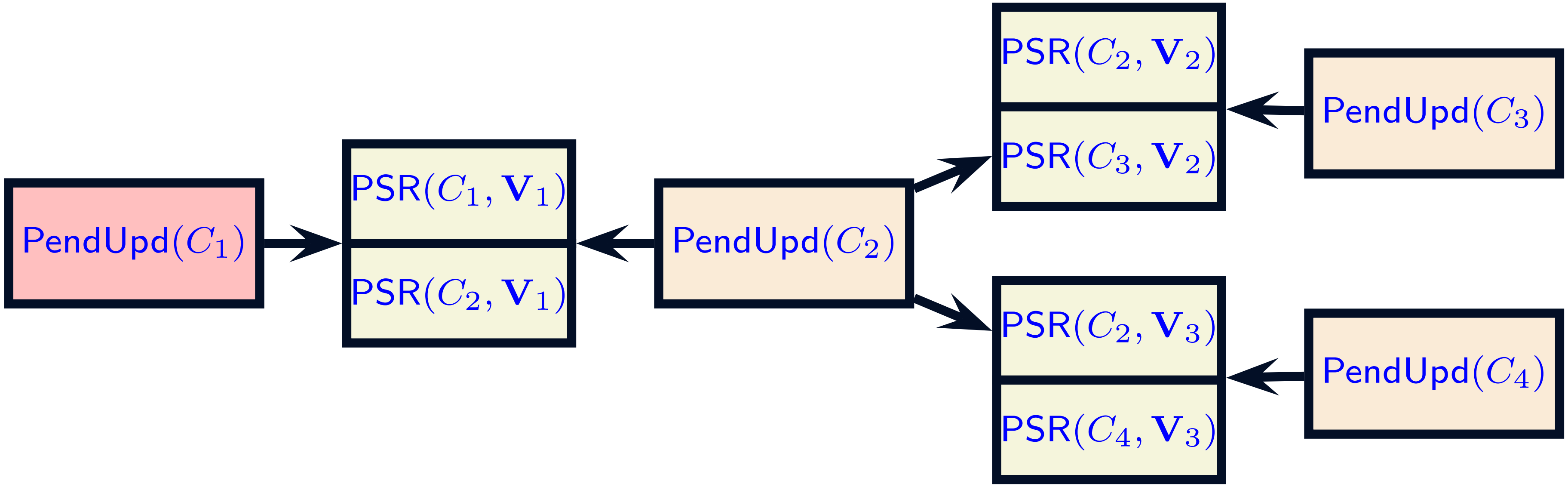
- To each component C_i corresponds a *pending-update register* PendUpd(C_i).
- To each view schema $\mathbf{V}_i$ is associated a *port-status register* PSR$(C_j, \mathbf{V}_i)$ for each component C_j which is connected to it.
- These additional registers are part of the control structure, and are in addition to the database itself.

Operation of Cooperative Update Management



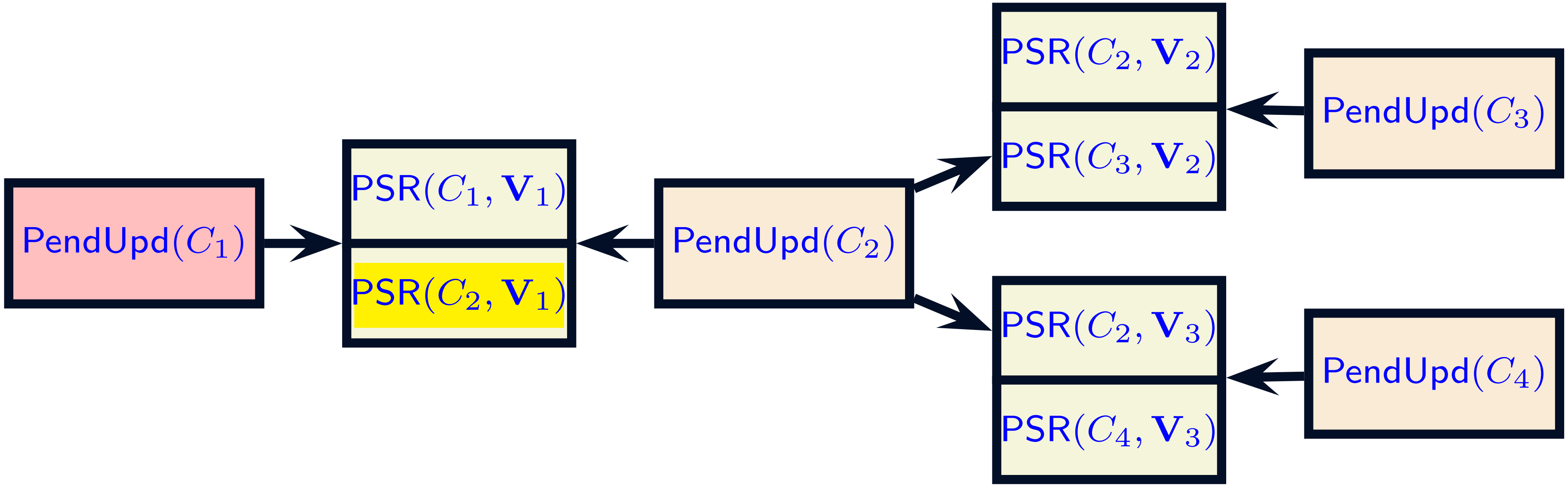
- All port-status registers are initially null.

Operation of Cooperative Update Management



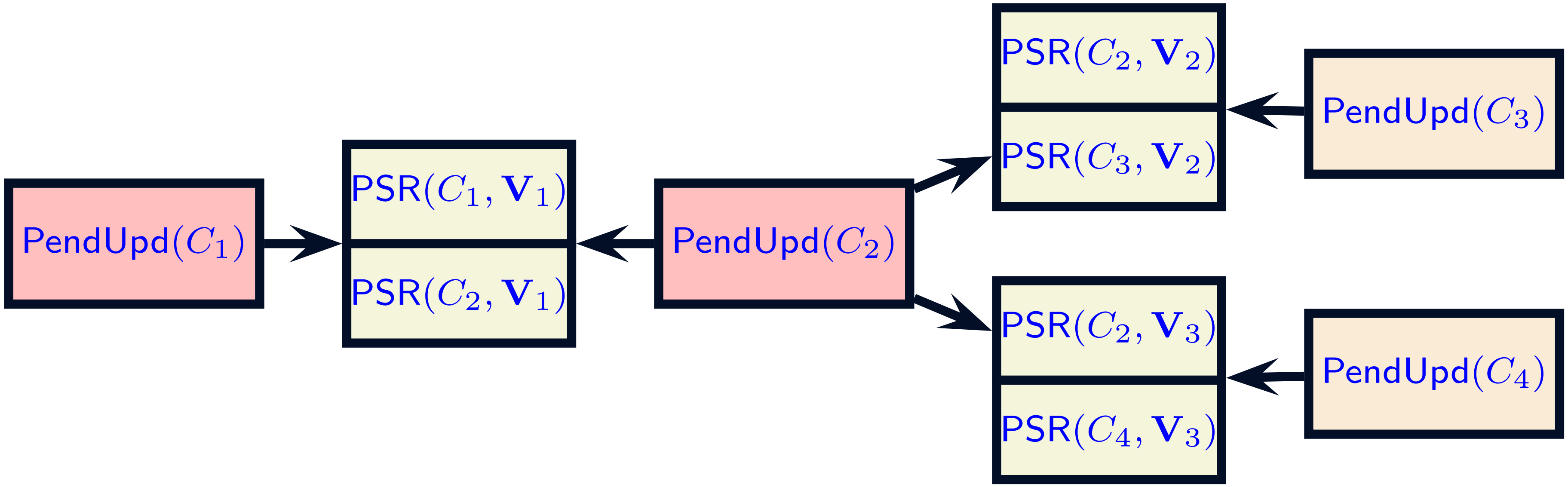
- All port-status registers are initially null.
- An update to a component schema

Operation of Cooperative Update Management



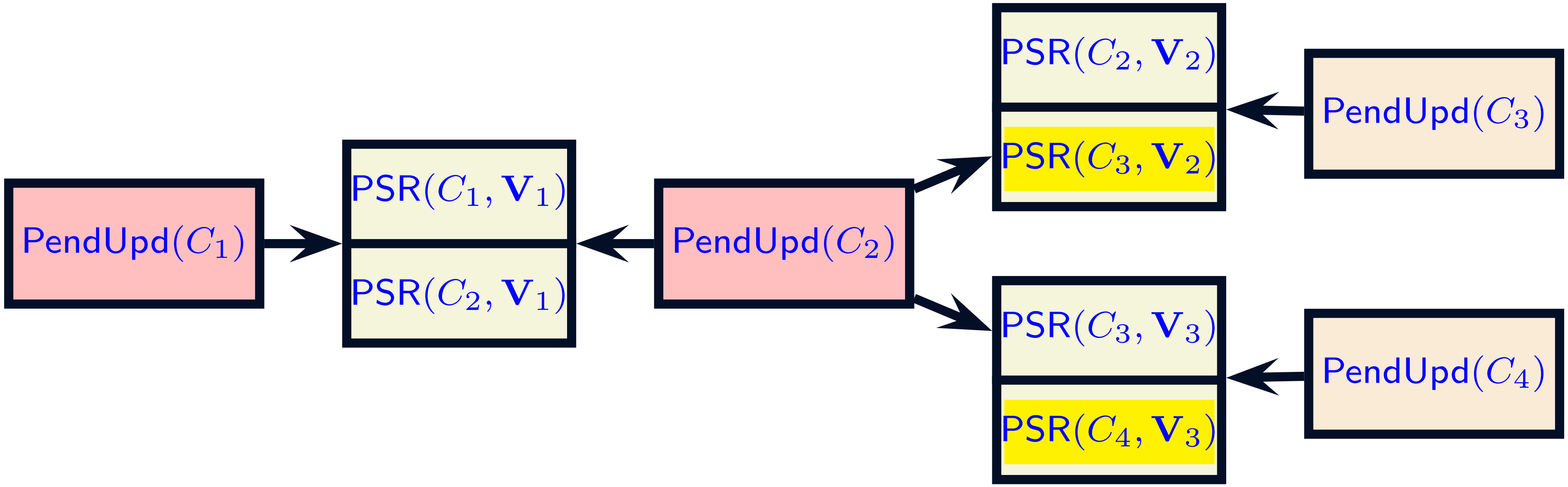
- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.

Operation of Cooperative Update Management



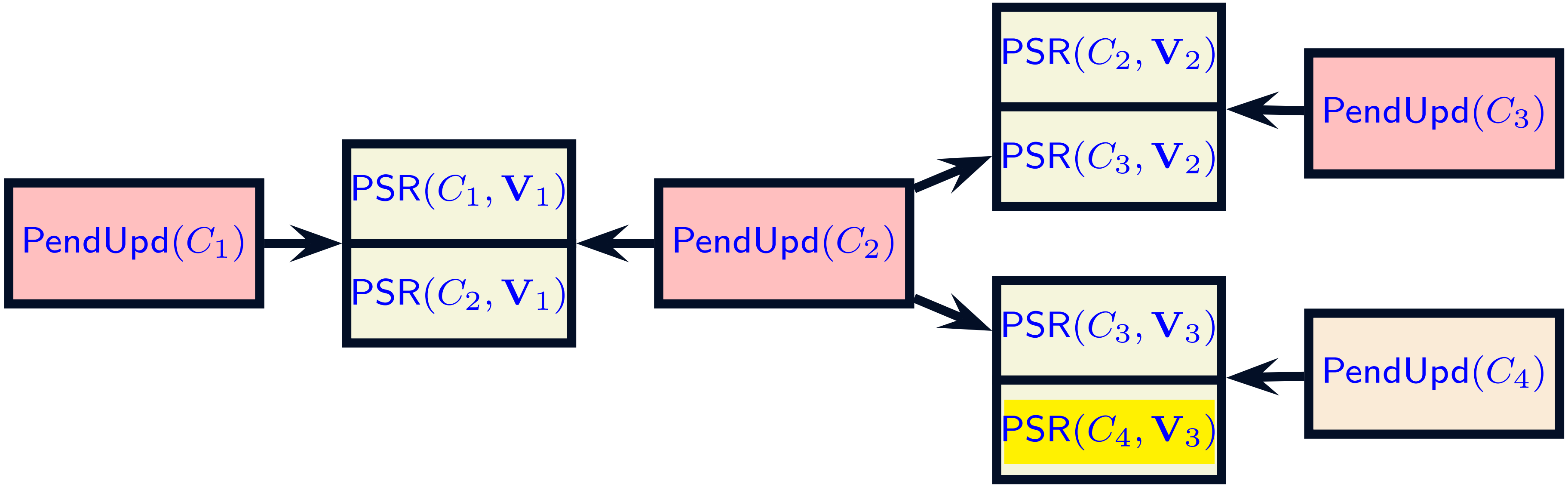
- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.

Operation of Cooperative Update Management

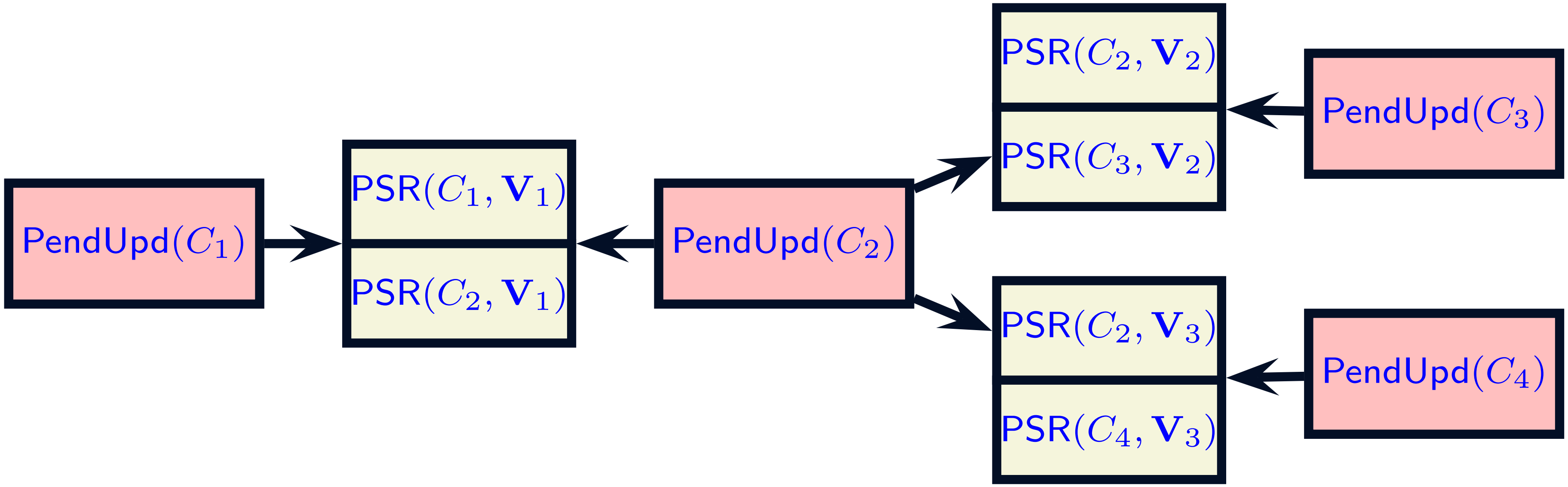


- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.

Operation of Cooperative Update Management

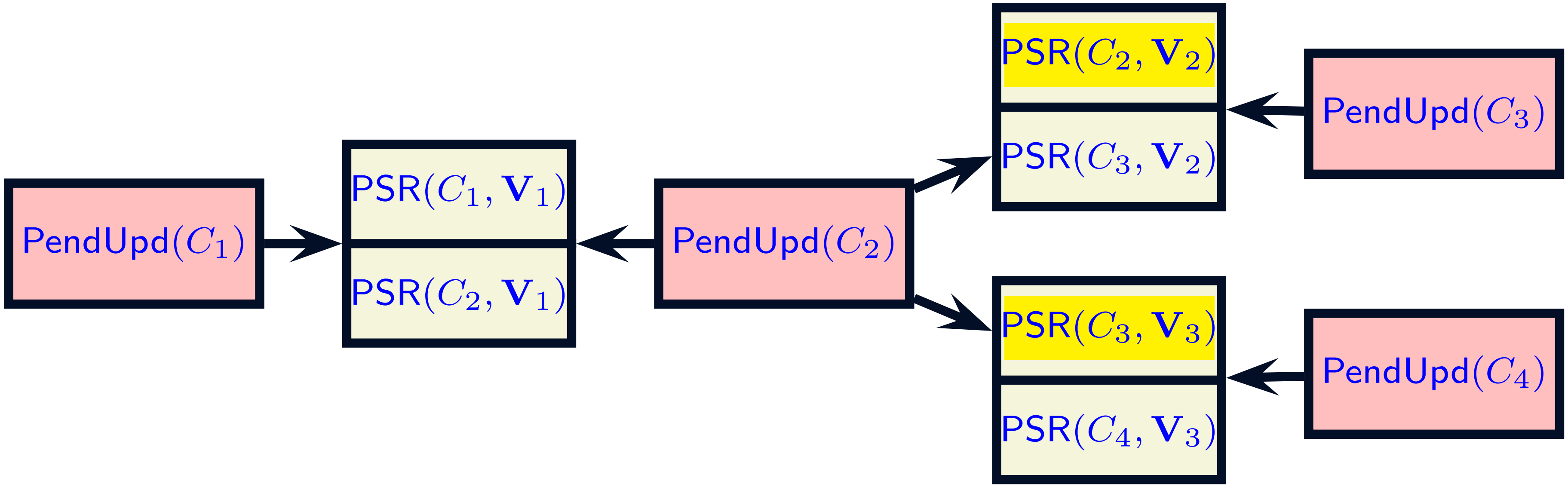


- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.



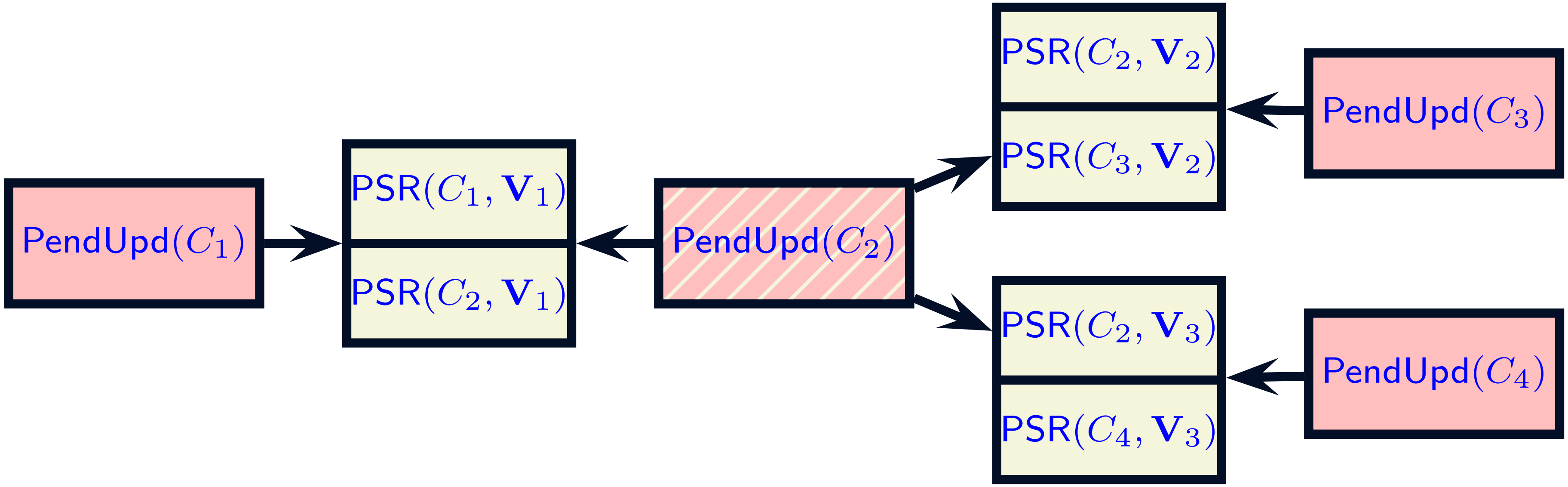
- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.

Operation of Cooperative Update Management



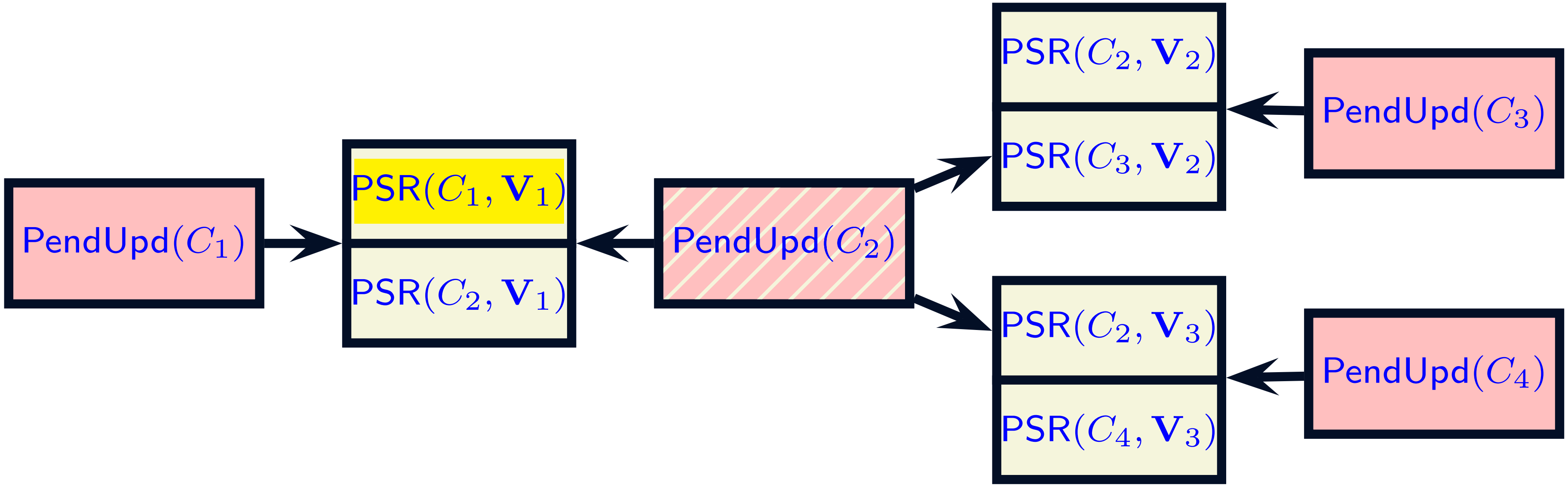
- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.

Operation of Cooperative Update Management



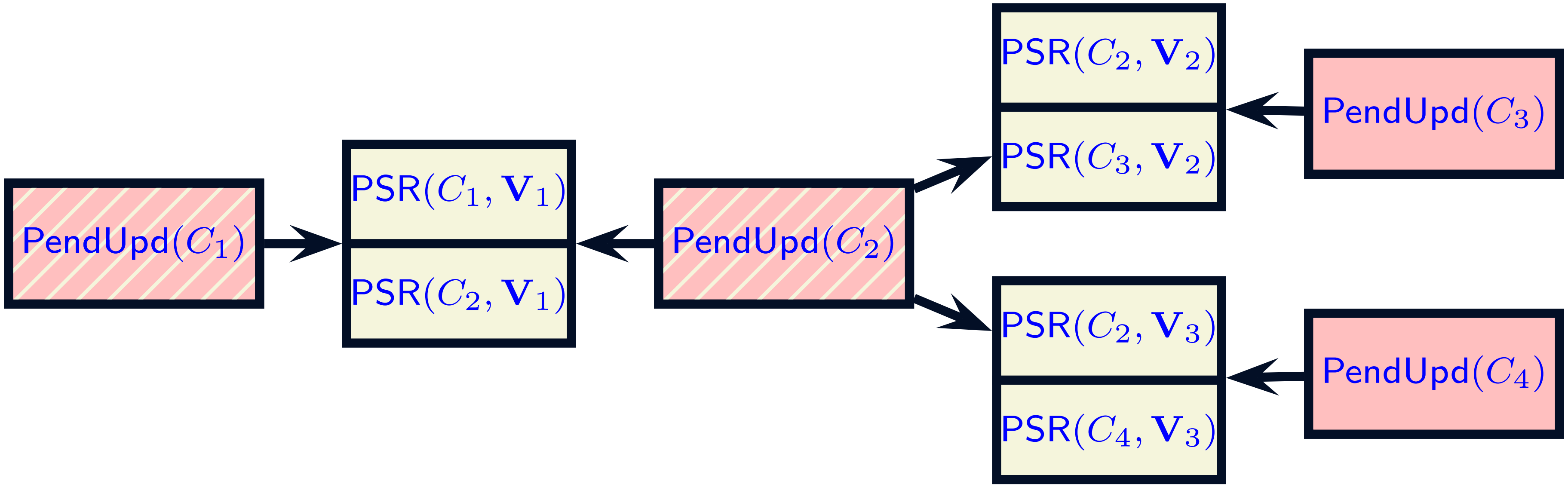
- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.

Operation of Cooperative Update Management



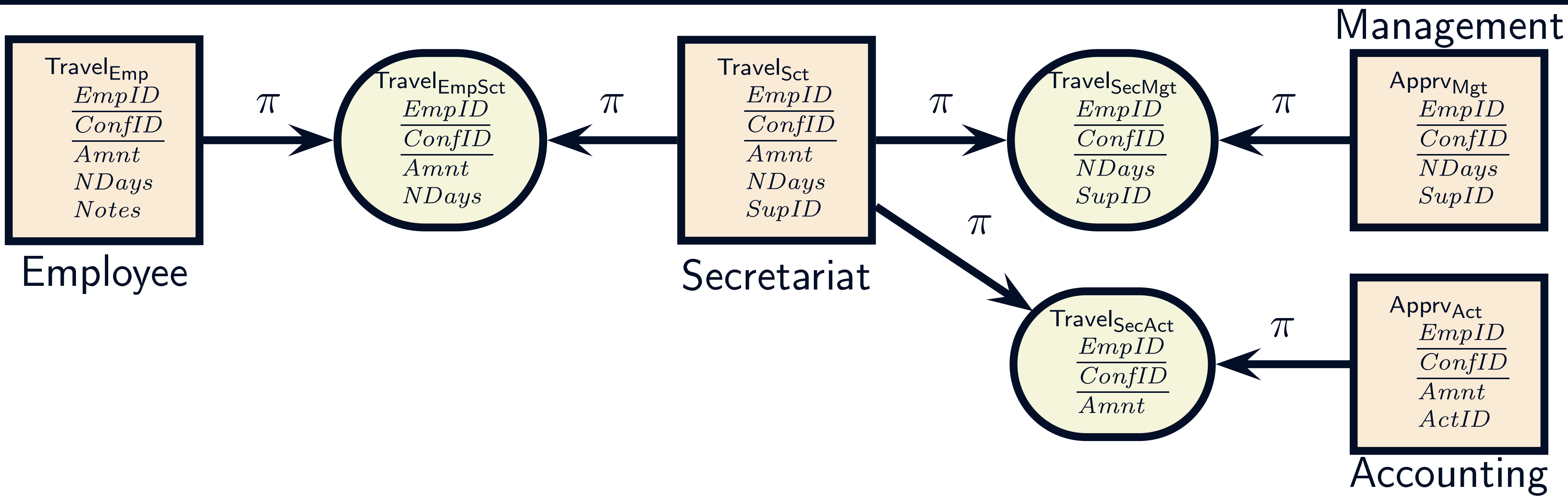
- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.

Operation of Cooperative Update Management

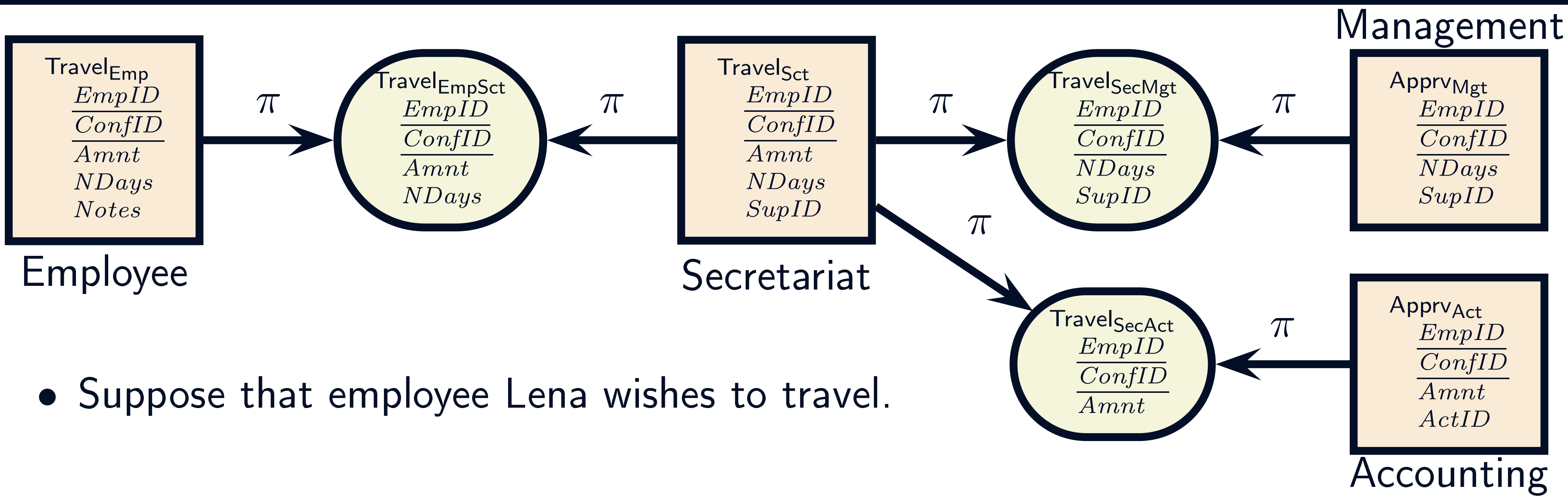


- All port-status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status register to null.
- The process repeats, and is nondeterministic.

Example: Travel Request and Authorization

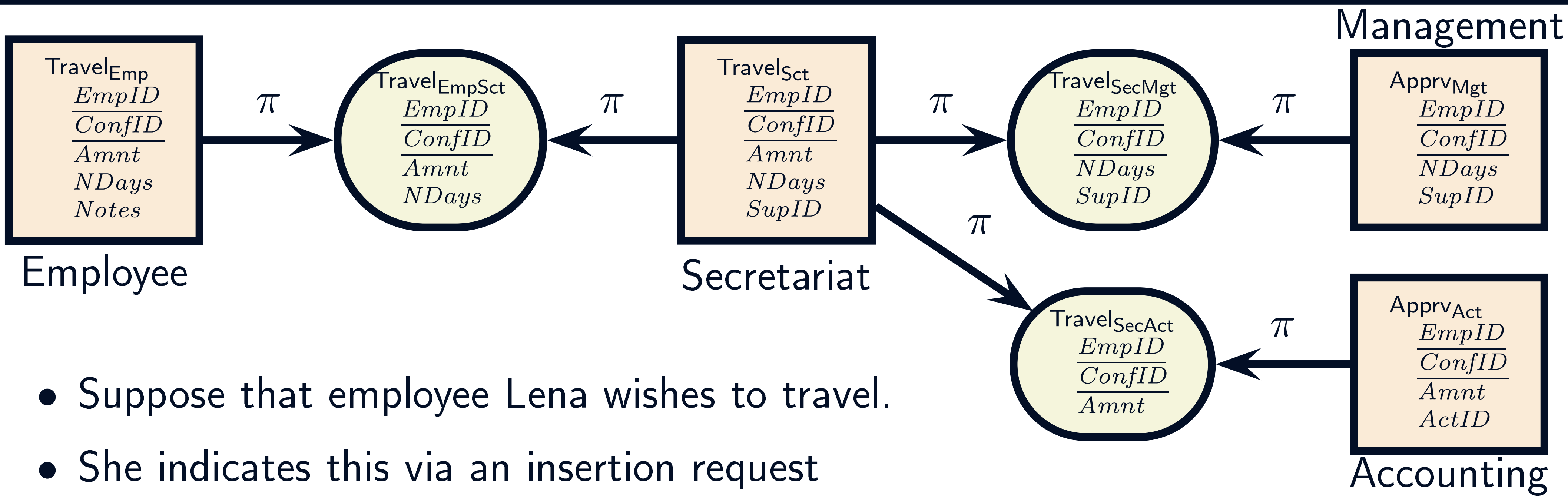


Example: Travel Request and Authorization



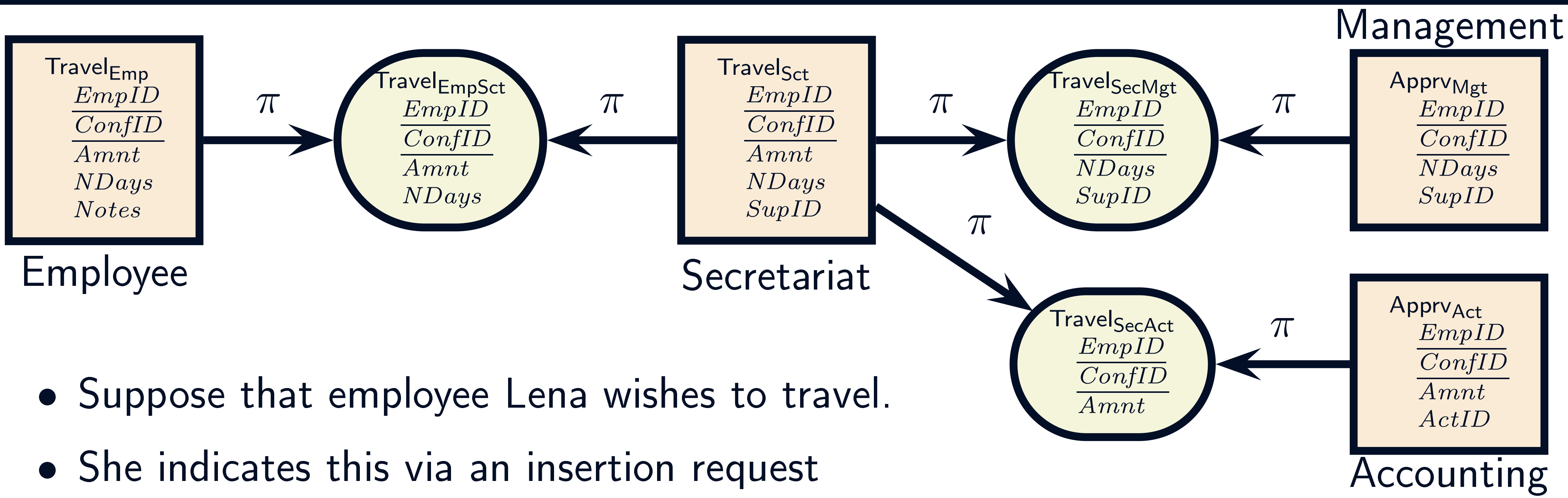
- Suppose that employee Lena wishes to travel.

Example: Travel Request and Authorization



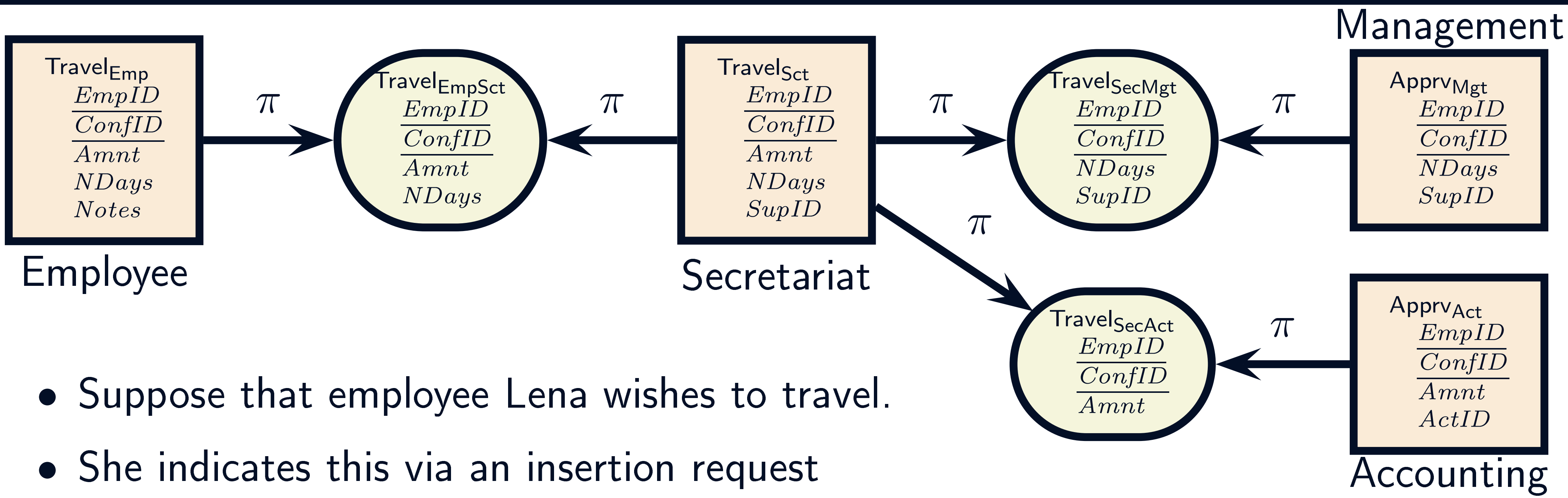
- Suppose that employee Lena wishes to travel.
- She indicates this via an insertion request into the schema of component Employee.

Example: Travel Request and Authorization



- Suppose that employee Lena wishes to travel.
- She indicates this via an insertion request into the schema of component Employee.
- Such a request is typically in the form of a finite *ranked set* of alternatives.

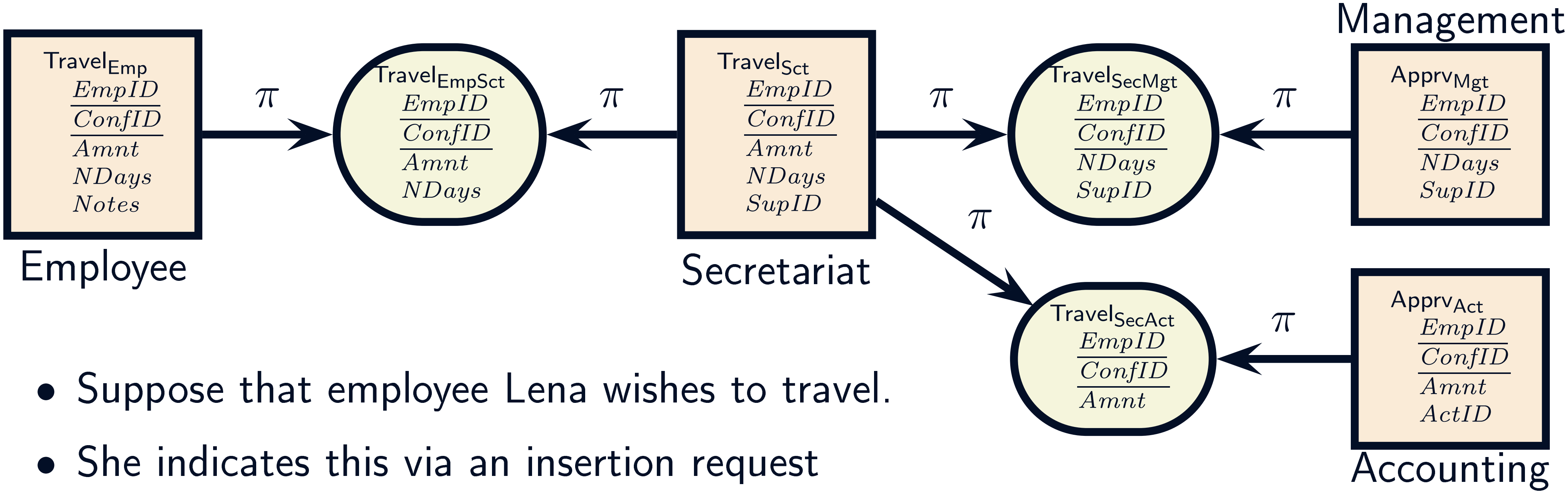
Example: Travel Request and Authorization



- Suppose that employee Lena wishes to travel.
- She indicates this via an insertion request into the schema of component Employee.
- Such a request is typically in the form of a finite *ranked set* of alternatives.

$$\{\text{Travel}_{\text{Emp}}[\text{Lena}, \text{ADBIS}, e_A, d_A, \mathfrak{n}] \mid €800 \leq e_A \leq €2000,\ 5 \leq d_A \leq 10\} \cup$$
$$\{\text{Travel}_{\text{Emp}}[\text{Lena}, \text{DEXA}, e_D, d_D, \mathfrak{n}] \mid €1000 \leq e_D \leq €2000,\ 3 \leq d_D \leq 10\}$$

Example: Travel Request and Authorization

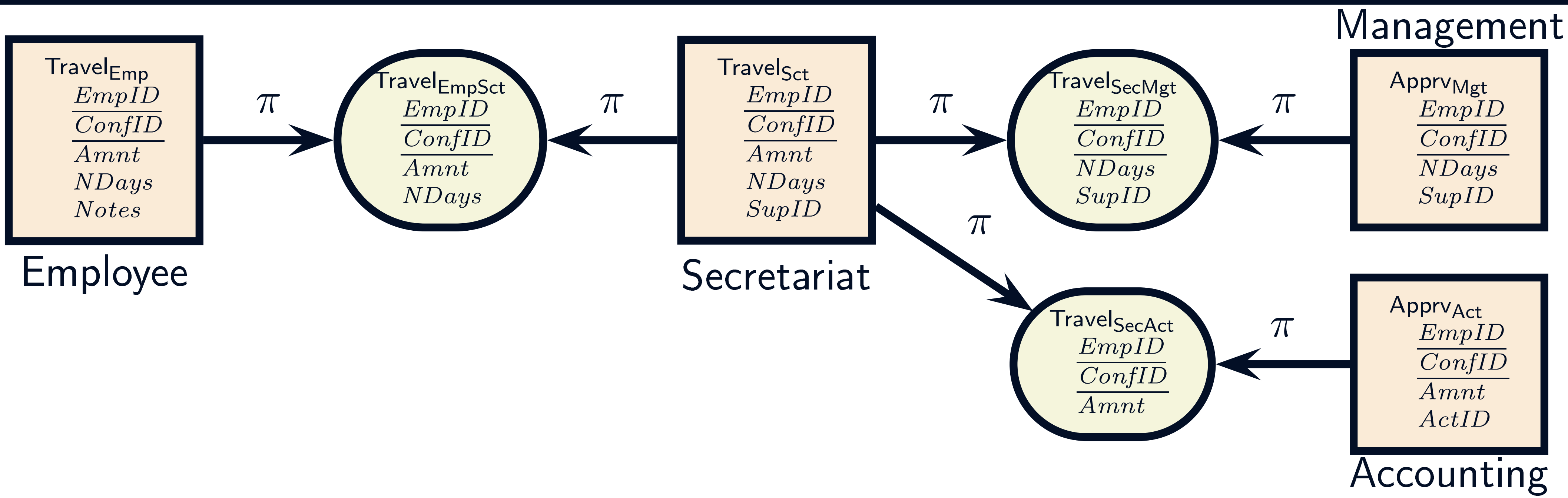


- Suppose that employee Lena wishes to travel.
- She indicates this via an insertion request into the schema of component Employee.
- Such a request is typically in the form of a finite *ranked set* of alternatives.

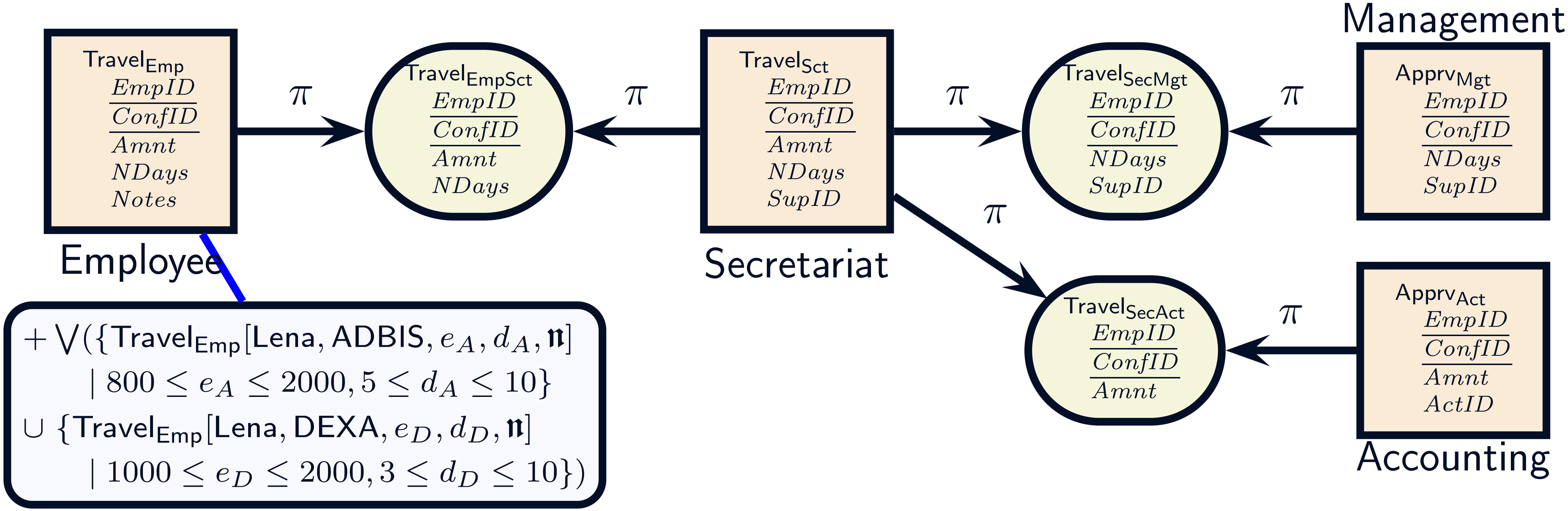
$$\{\text{Travel}_{\text{Emp}}[\text{Lena}, \text{ADBIS}, e_A, d_A, \mathfrak{n}] \mid €800 \leq e_A \leq €2000,\ 5 \leq d_A \leq 10\} \cup$$
$$\{\text{Travel}_{\text{Emp}}[\text{Lena}, \text{DEXA}, e_D, d_D, \mathfrak{n}] \mid €1000 \leq e_D \leq €2000,\ 3 \leq d_D \leq 10\}$$

  - ADBIS is always preferred to DEXA.
  - For a given conference, more money and days are always preferred to fewer.

Example: Evolution of Travel Request and Authorization



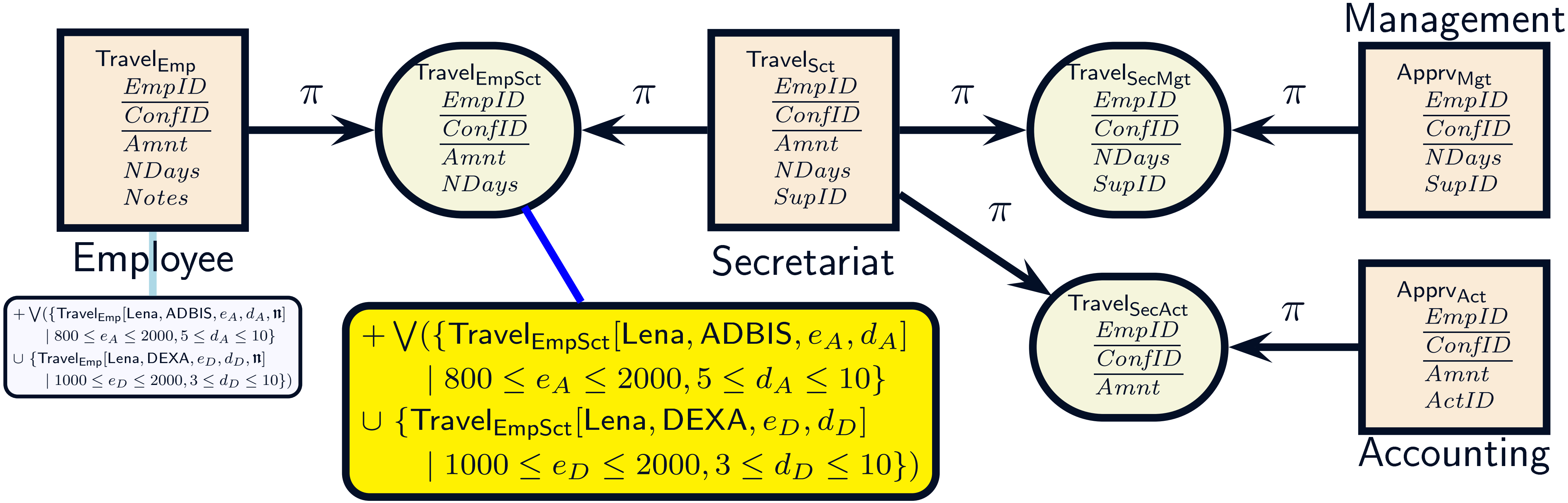
- The evolution of a specific update request will now be illustrated.



$$
\begin{aligned}
&+\bigvee(\{\mathsf{Travel}_{\mathsf{Emp}}[\mathsf{Lena}, \mathsf{ADBIS}, e_A, d_A, \mathfrak{n}] \\
&\qquad \mid 800 \leq e_A \leq 2000, 5 \leq d_A \leq 10\} \\
&\cup \{\mathsf{Travel}_{\mathsf{Emp}}[\mathsf{Lena}, \mathsf{DEXA}, e_D, d_D, \mathfrak{n}] \\
&\qquad \mid 1000 \leq e_D \leq 2000, 3 \leq d_D \leq 10\})
\end{aligned}
$$

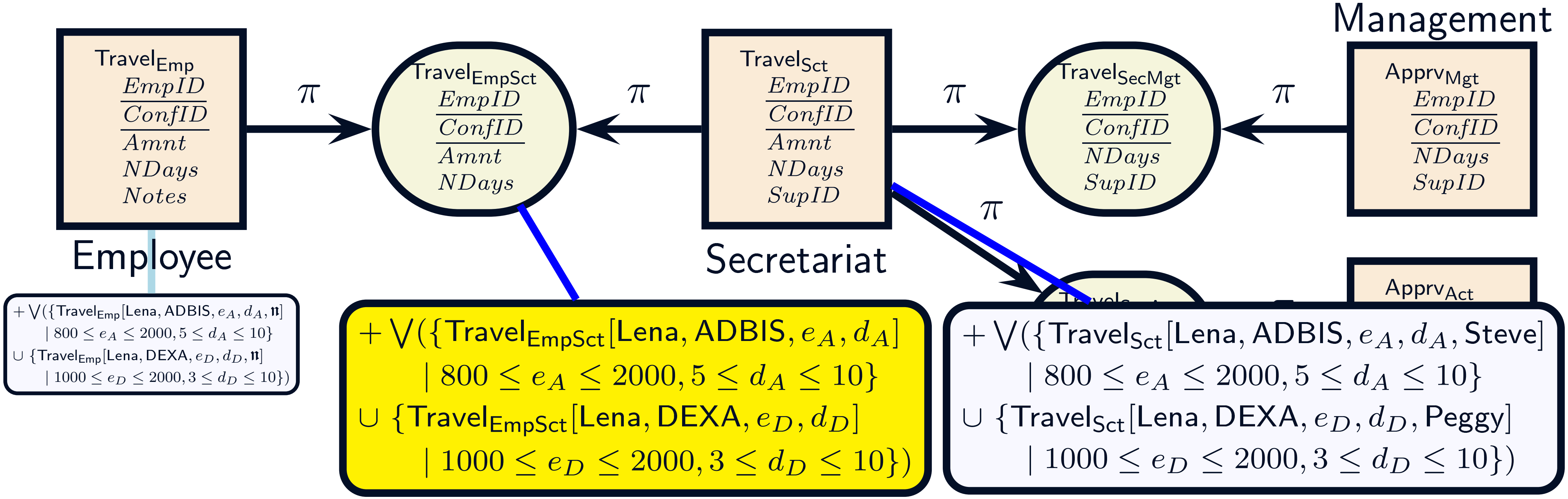
- First, the desired ranked update is entered into the pending-update register for Employee. Notation:
  - $+$ = Insert.
  - $\bigvee$ = Choose one of the alternatives.

Example: Evolution of Travel Request and Authorization



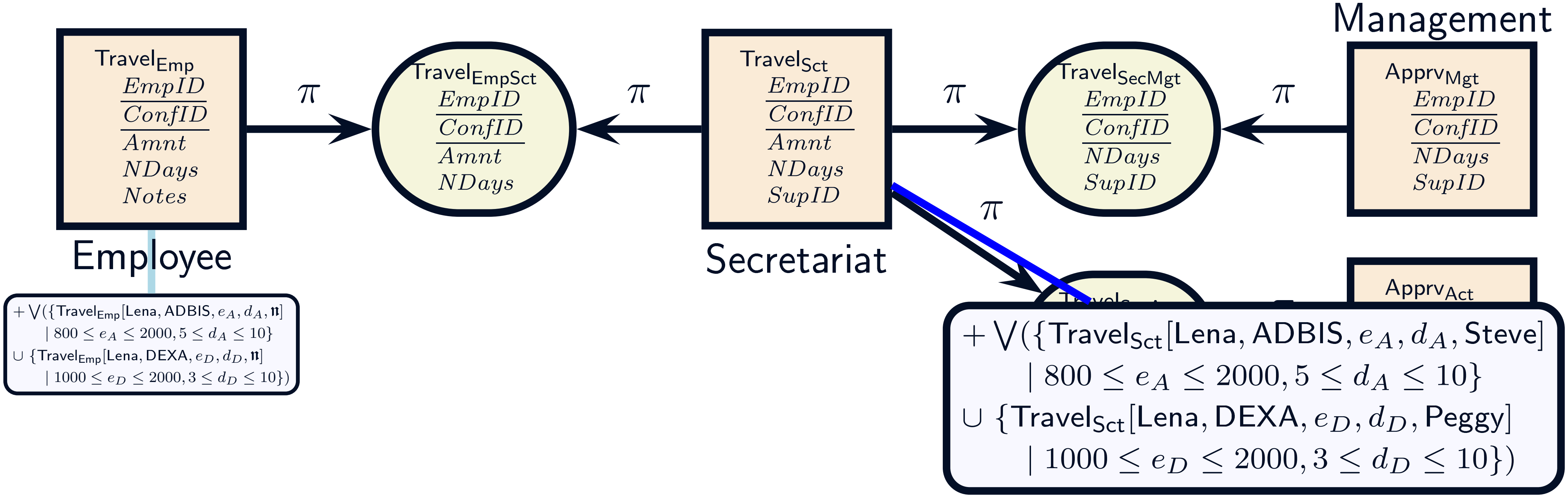
- This update is then projected to the port-status register which connects Employee to Secretariat.

Example: Evolution of Travel Request and Authorization



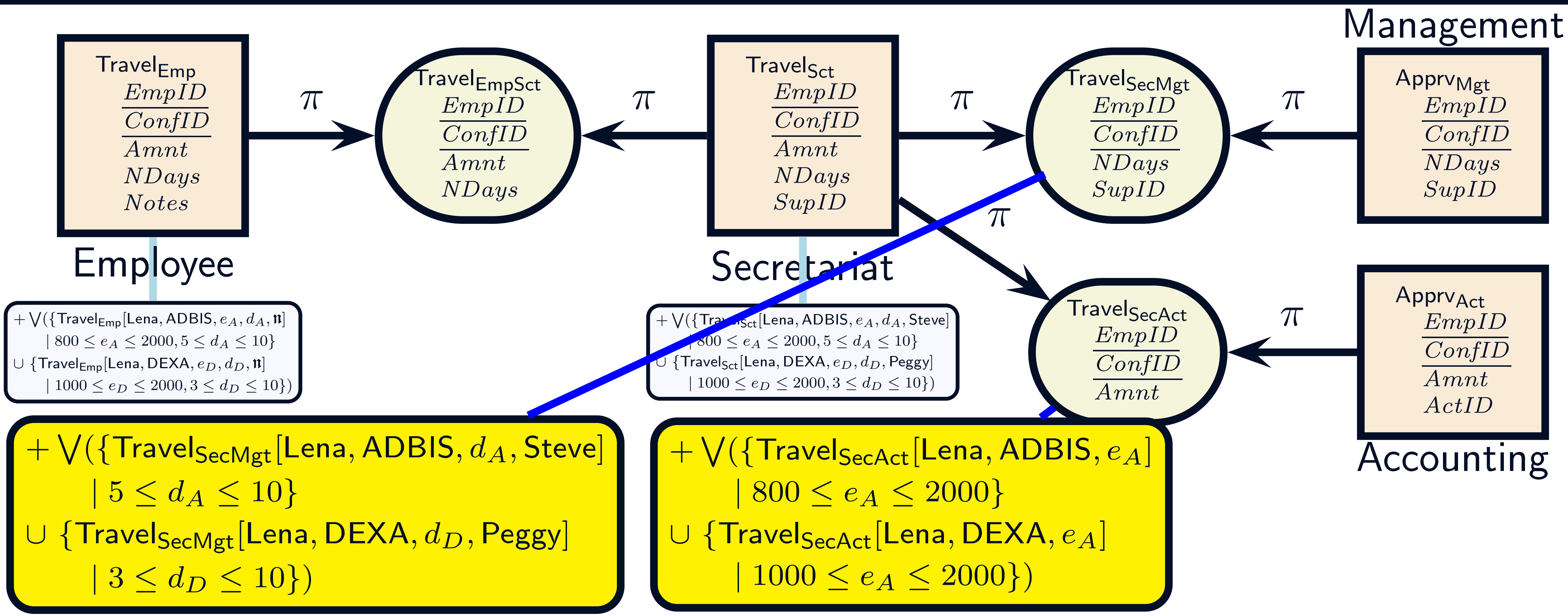
- The user of the Secretariat component then lifts this update to one on that component. It is placed in the pending-update register for that component.
  - Note that decisions must be made.
  - One of many possible liftings must be selected.

Example: Evolution of Travel Request and Authorization



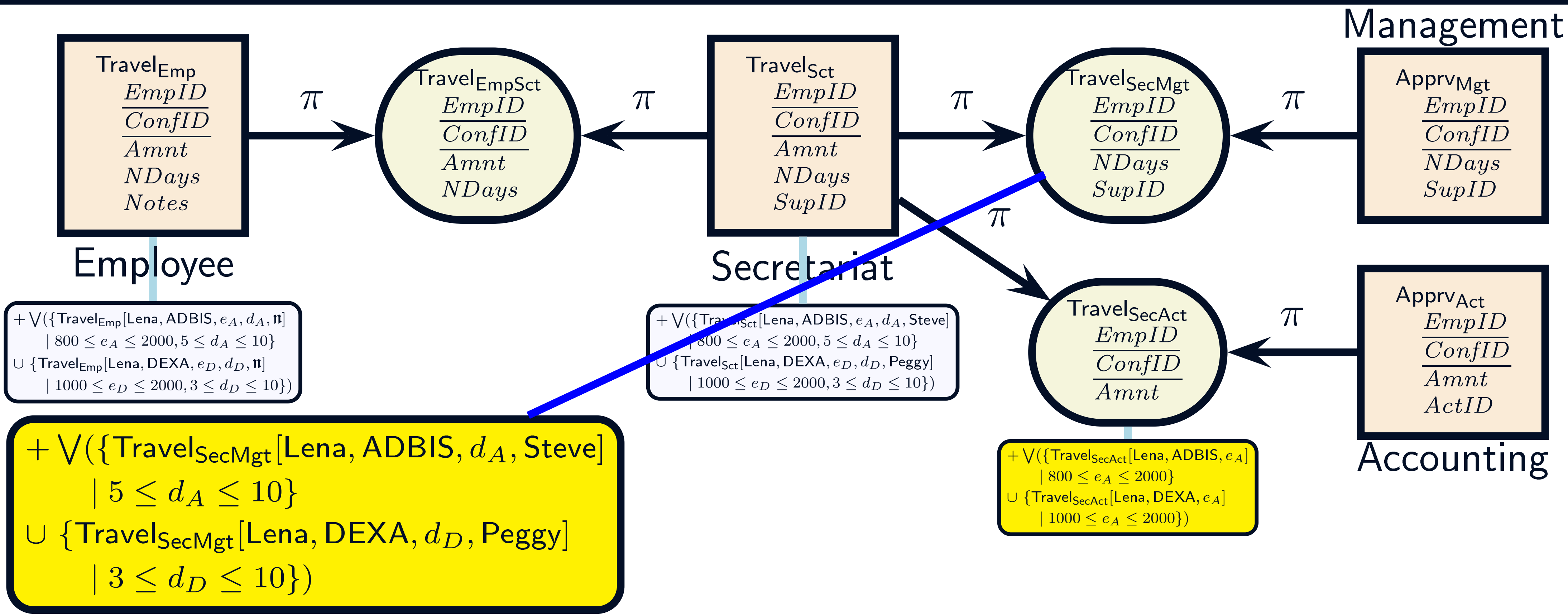
- The port-status register is then cleared, since this update has been processed.

Example: Evolution of Travel Request and Authorization



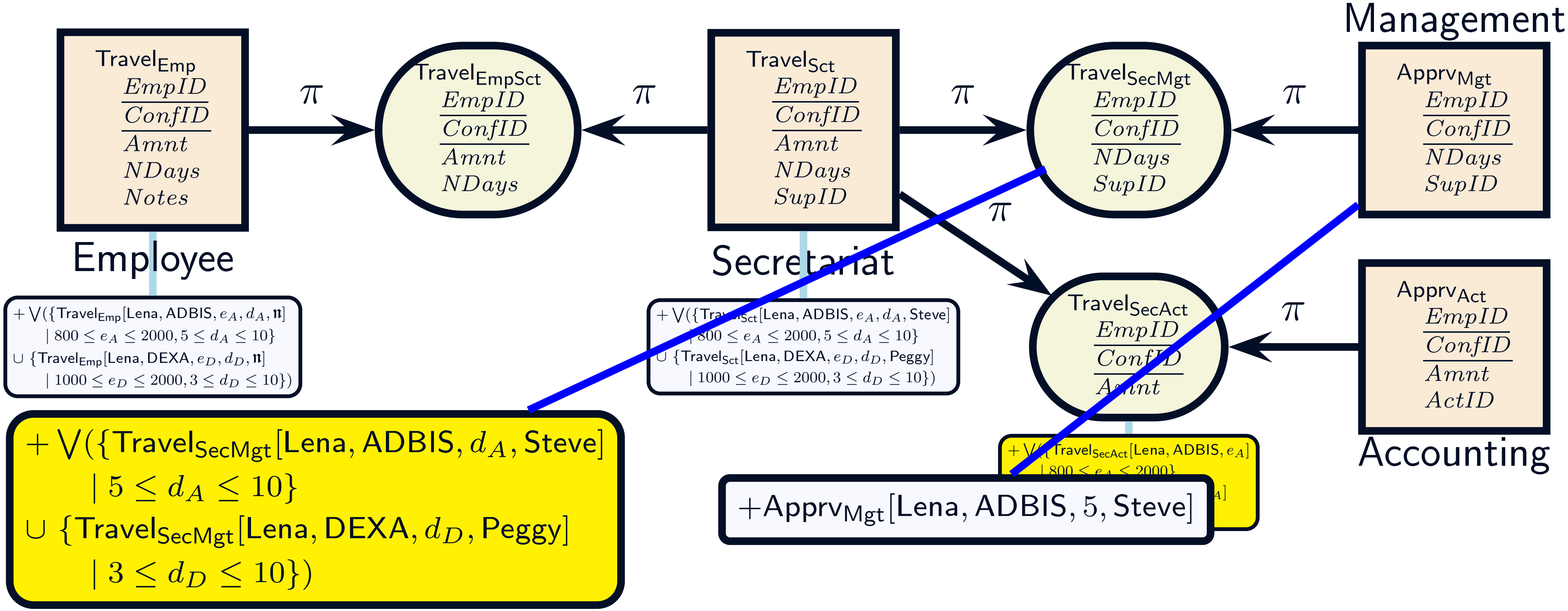
- This lifted update is then projected into the appropriate port-status registers which connect Secretariat to Management and Accounting.
- It is not projected back onto the port-status register which is connected to Employee, because the new value would be the same as the old one.

Example: Evolution of Travel Request and Authorization



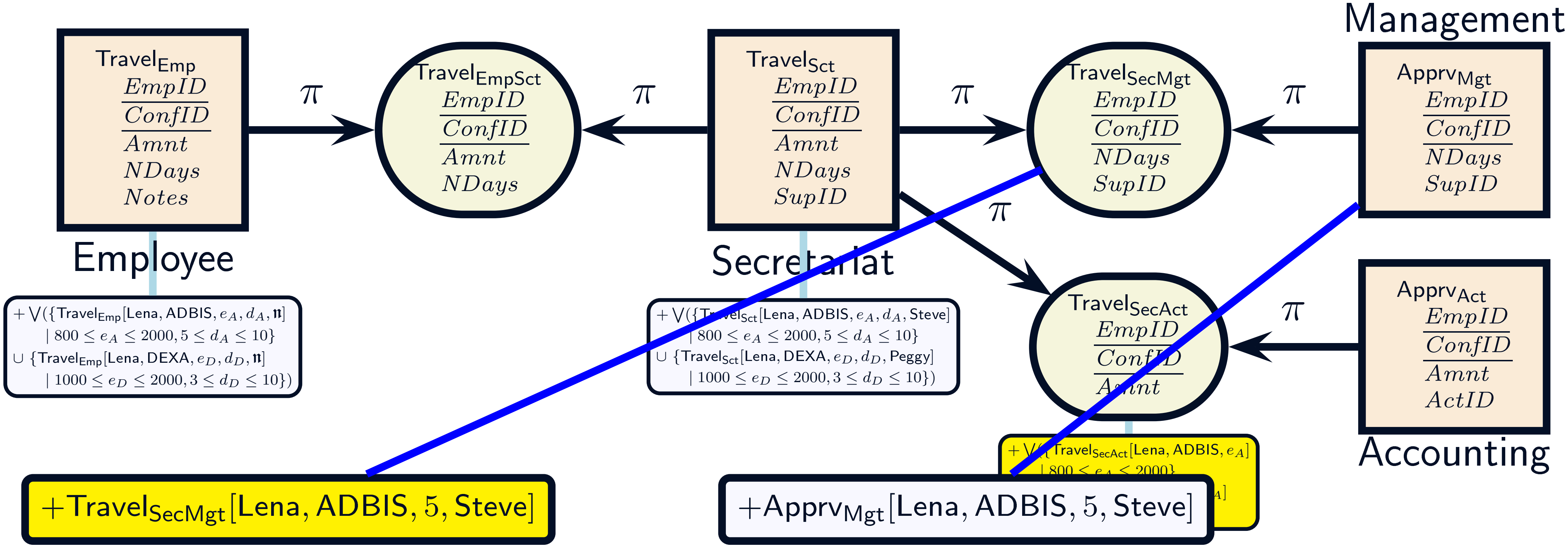
- First consider lifting the projected update to the Management component.
    - Again, there are decisions to be made.

Example: Evolution of Travel Request and Authorization

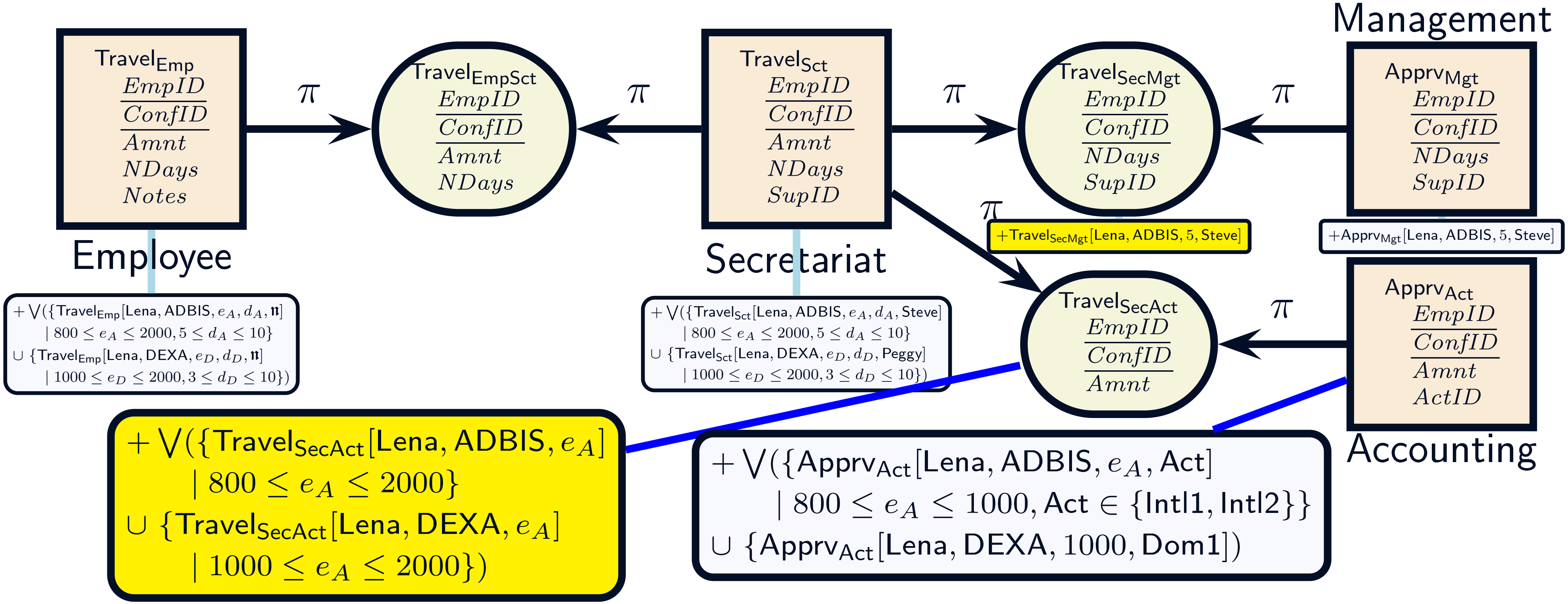


- Manager Steve processes the request, and decides to allow Lena to attend ADBIS for five days.

Example: Evolution of Travel Request and Authorization

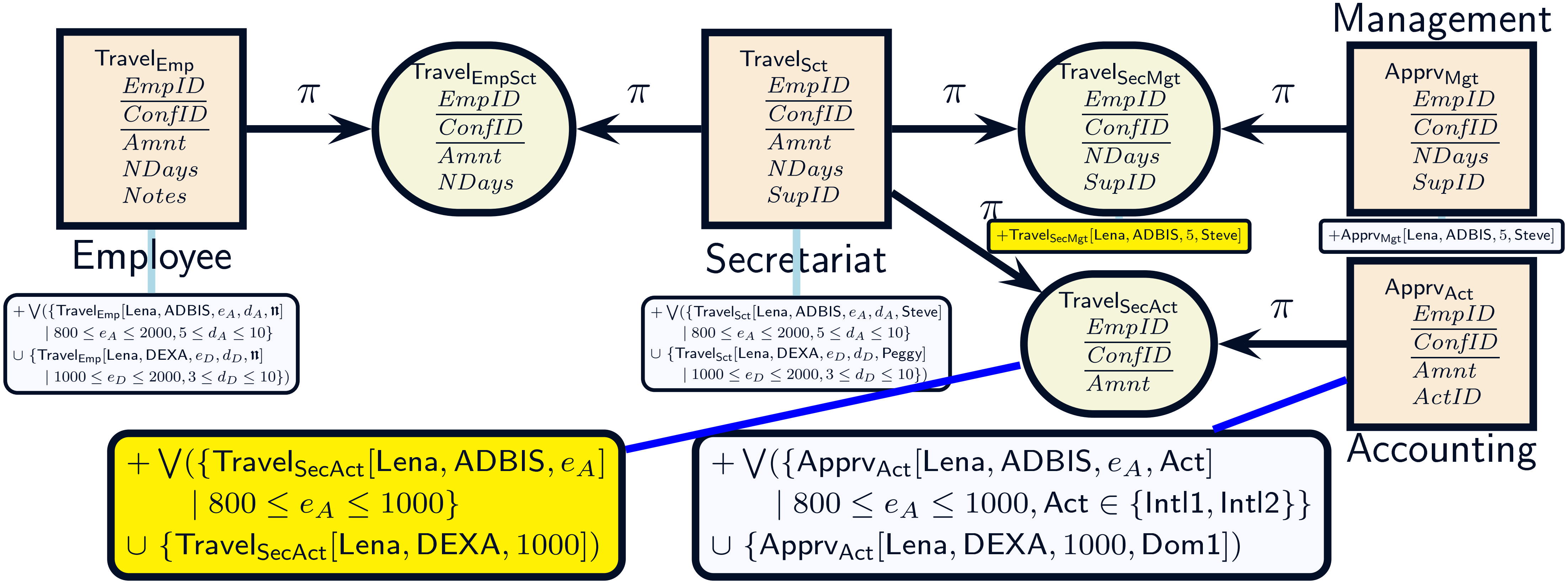


- The value in the port-status register for the Management component is removed, but a new value for the port-status register for Secretariat is inserted.



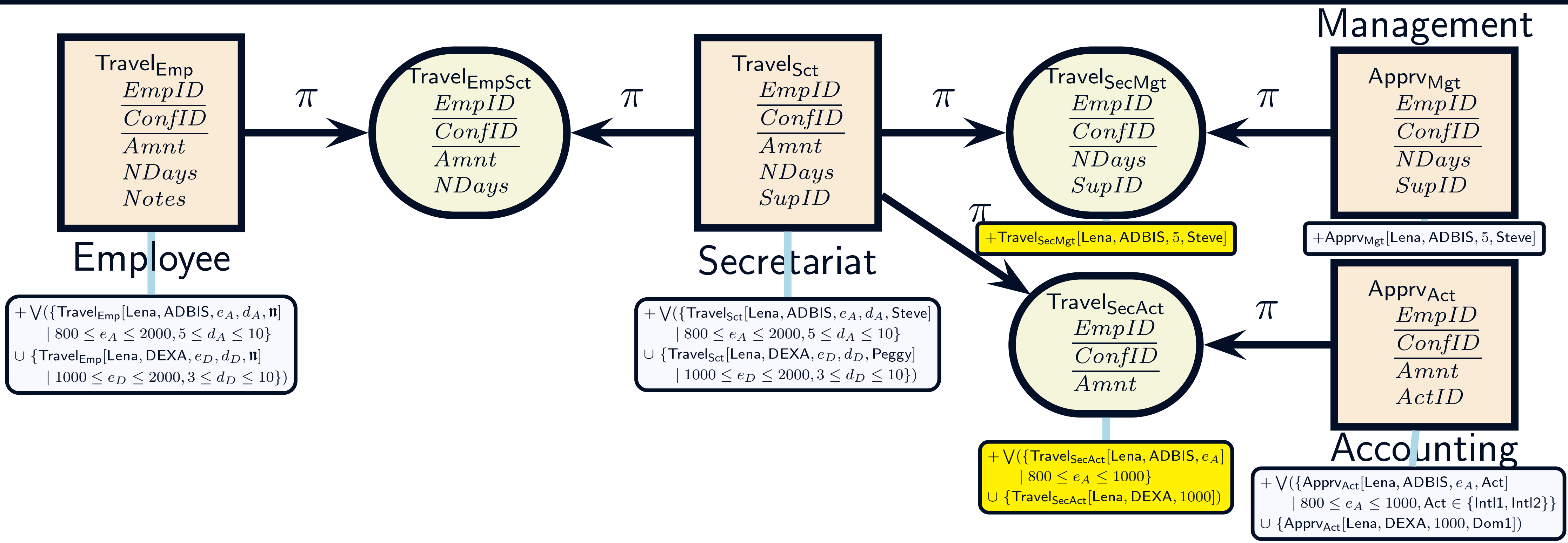
- Now consider lifting the projected update to the Management component.
  - The accounting manager decides to award €1000 for the requested travel.
  - The appropriate accounts are also identified.

Example: Evolution of Travel Request and Authorization



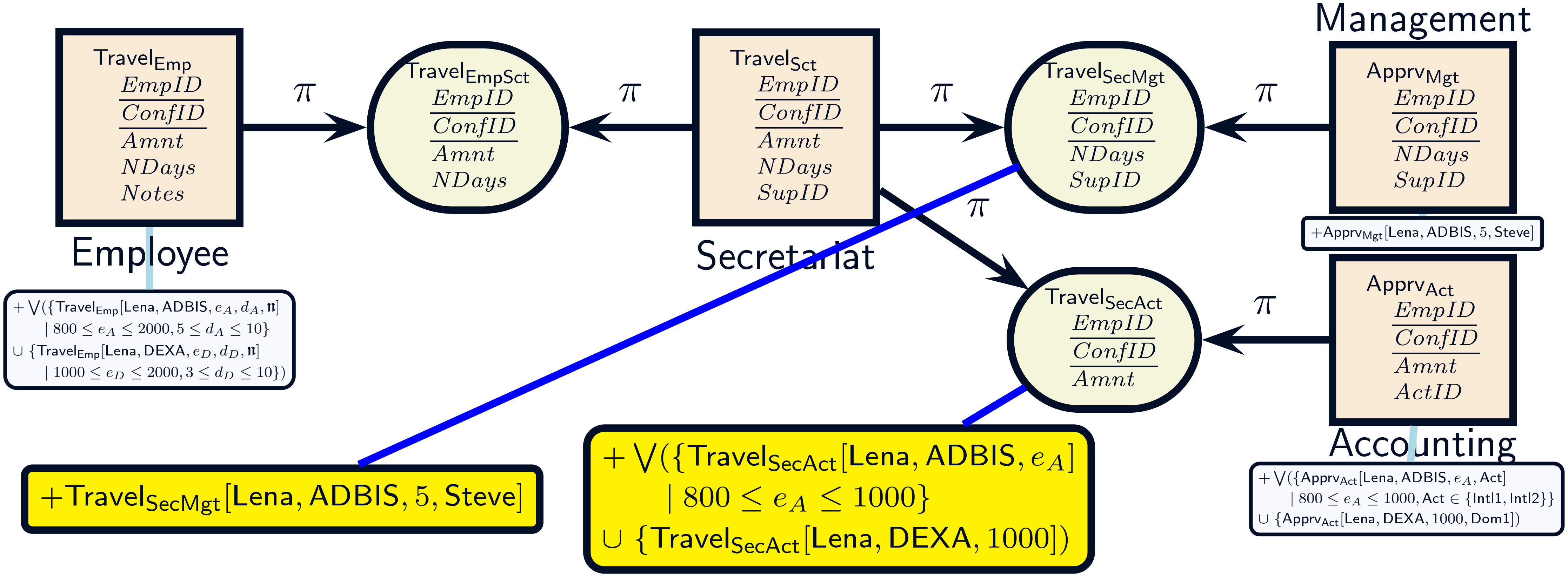
- The value in the port-status register for the Accounting component is cleared, but a new value for the port-status register for Secretariat is inserted.

Example: Evolution of Travel Request and Authorization



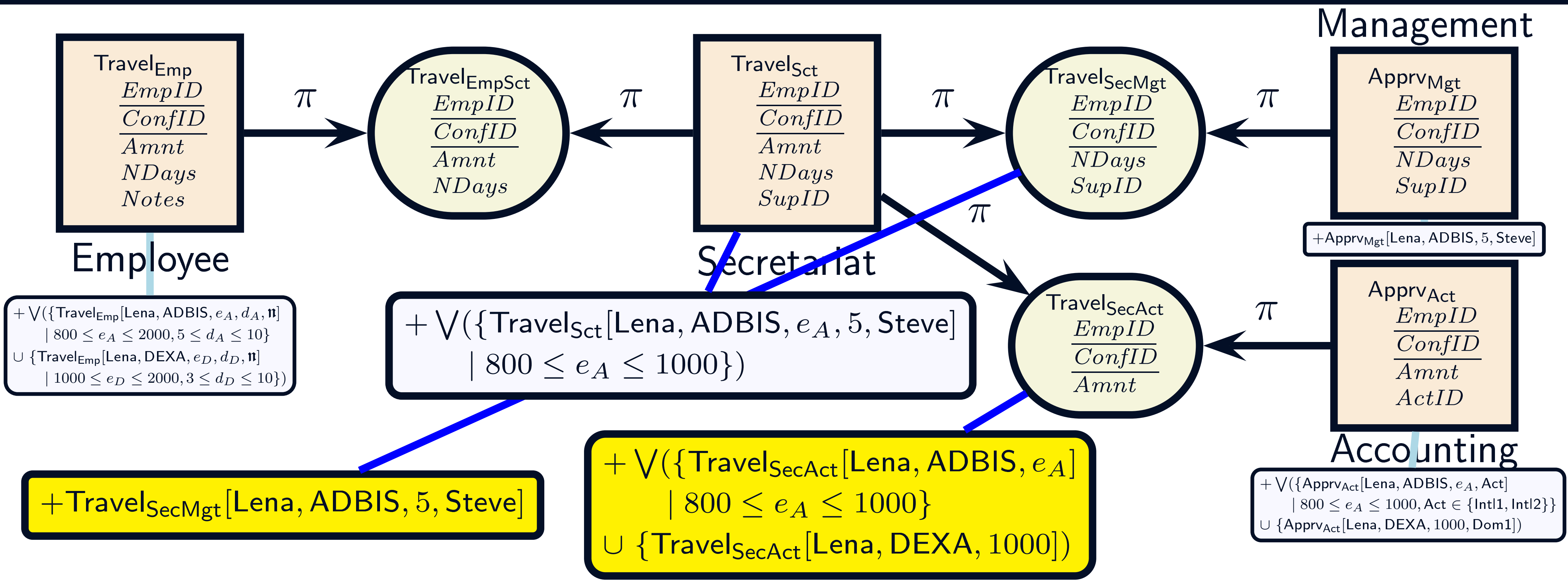
- Now the update negotiation propagates back right to left.

Example: Evolution of Travel Request and Authorization



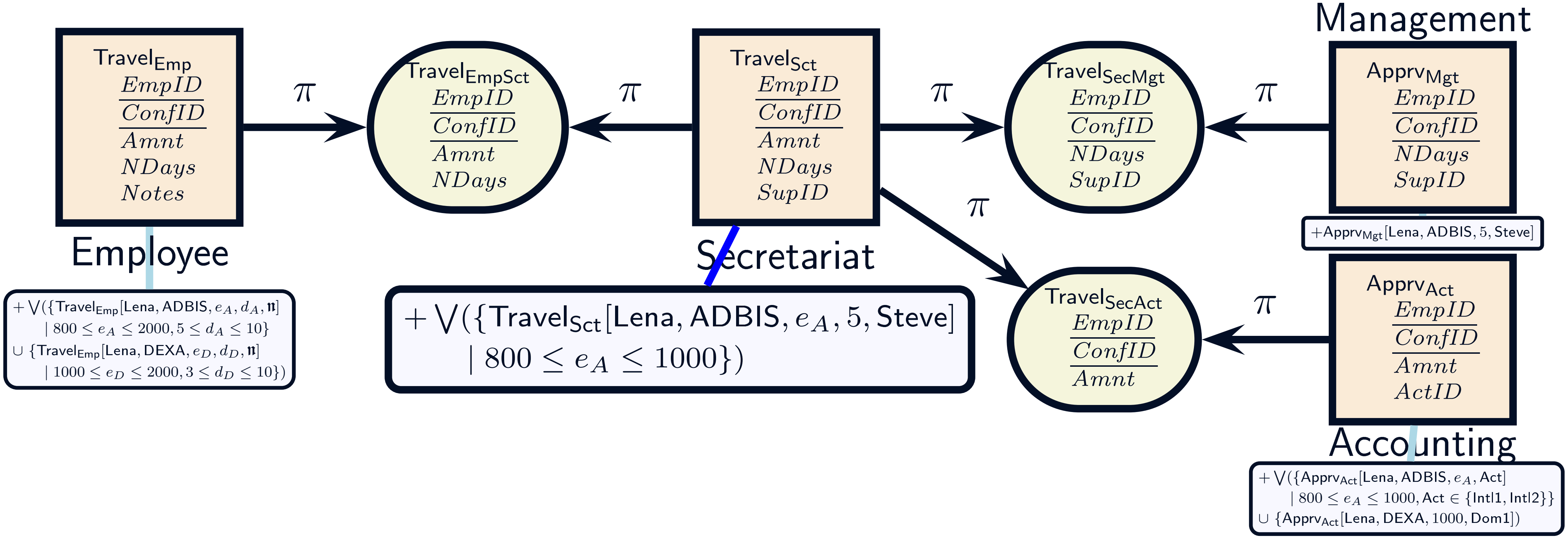
- The two values in the port-status registers must be lifted to the Secretariat component simultaneously.

Example: Evolution of Travel Request and Authorization

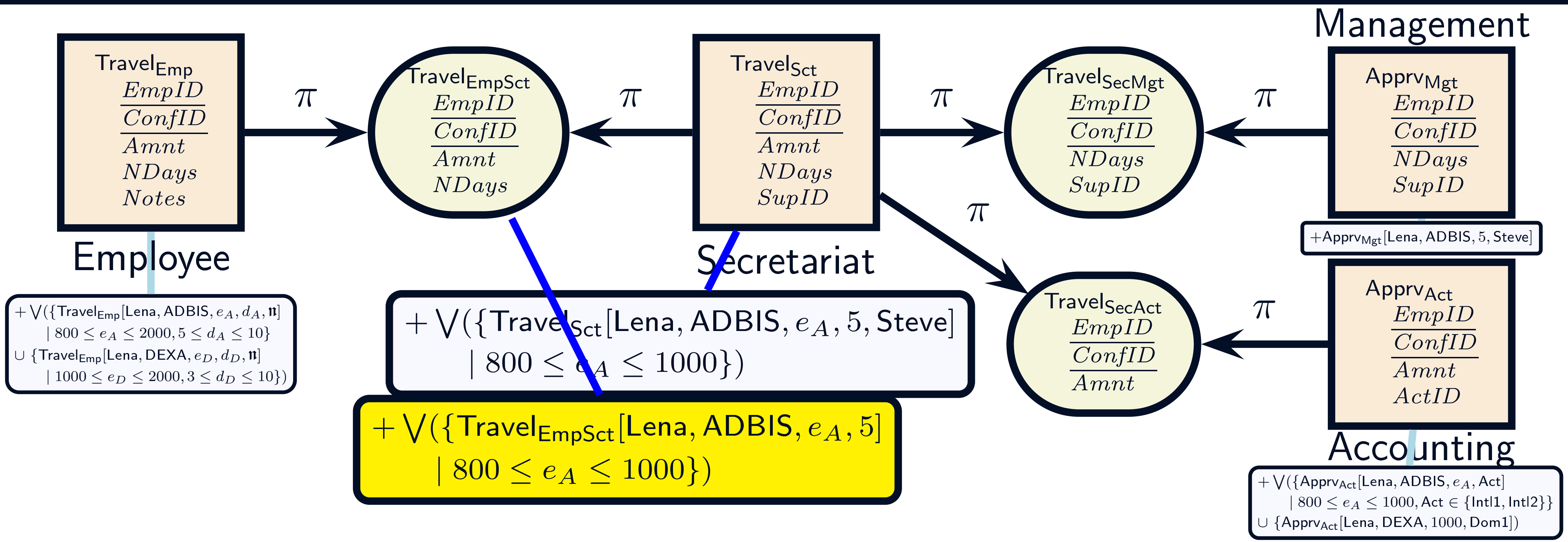


- The maximal lifting is selected.
  - The Secretariat imposes no additional limitations.

Example: Evolution of Travel Request and Authorization

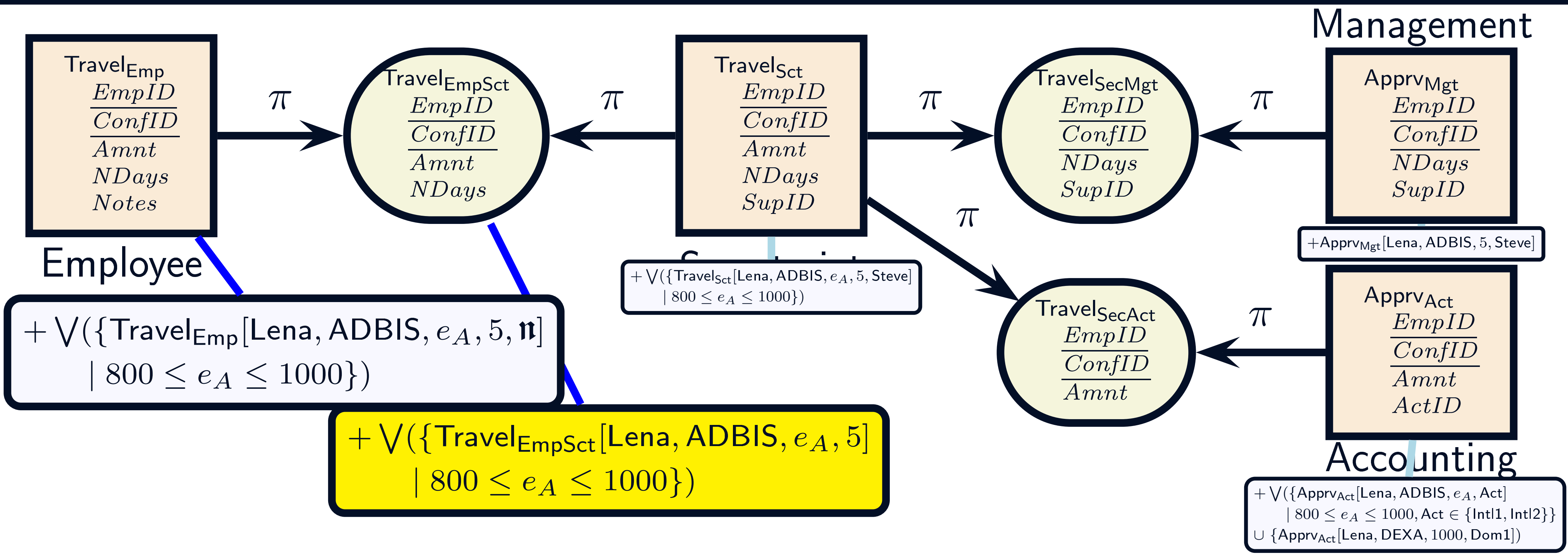


- The two port-status registers are now cleared.



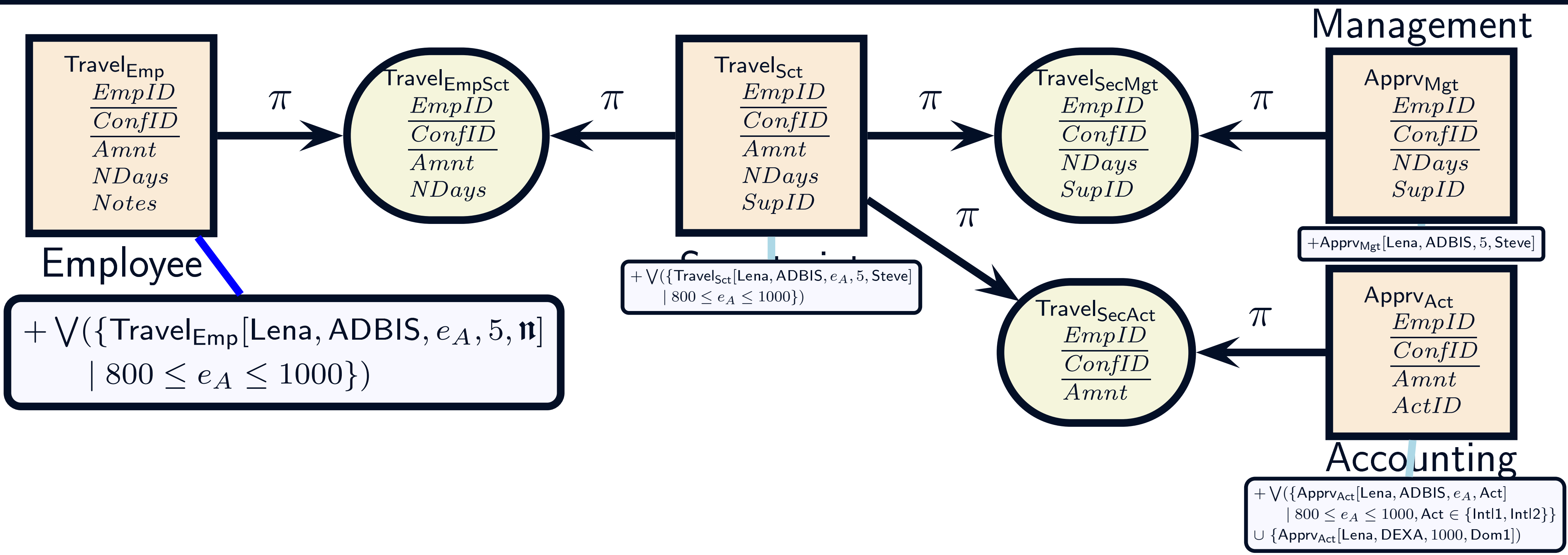
- The update request is then projected to the appropriate port-status register connecting Secretariat to Employee.

Example: Evolution of Travel Request and Authorization



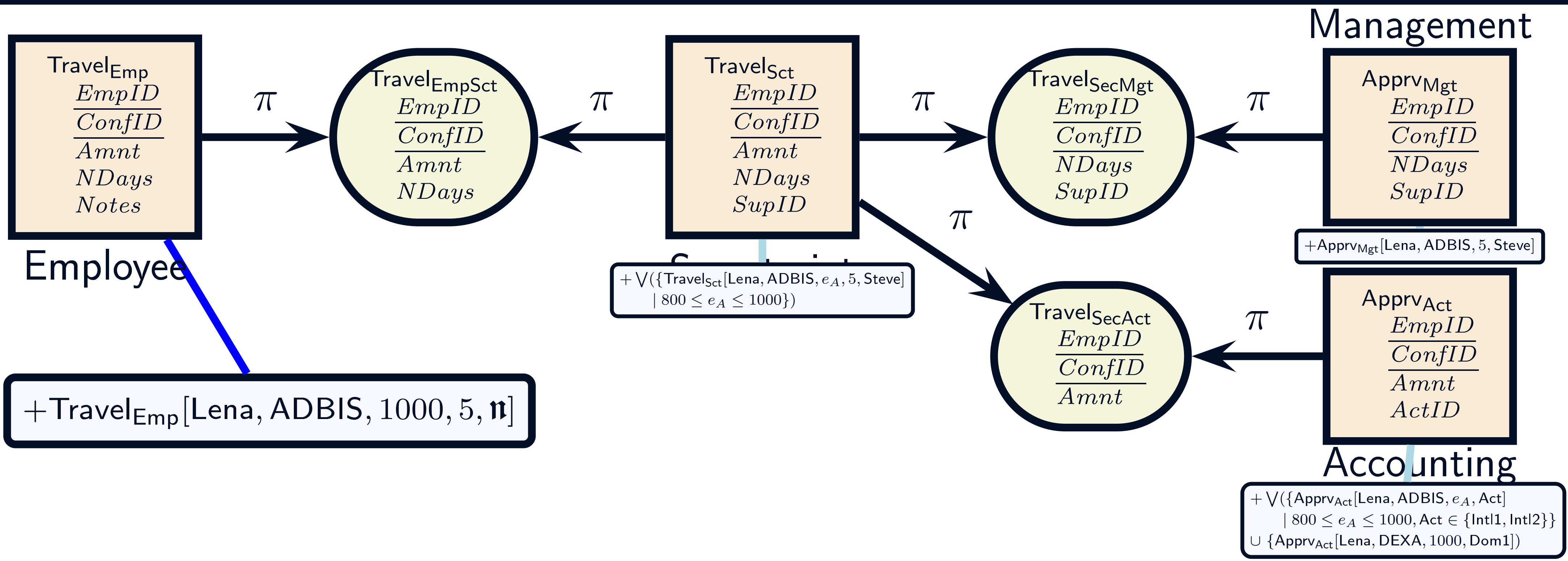
- This update request is then lifted to the Employee component.
    - Again, the maximal lifting is selected.

Example: Evolution of Travel Request and Authorization



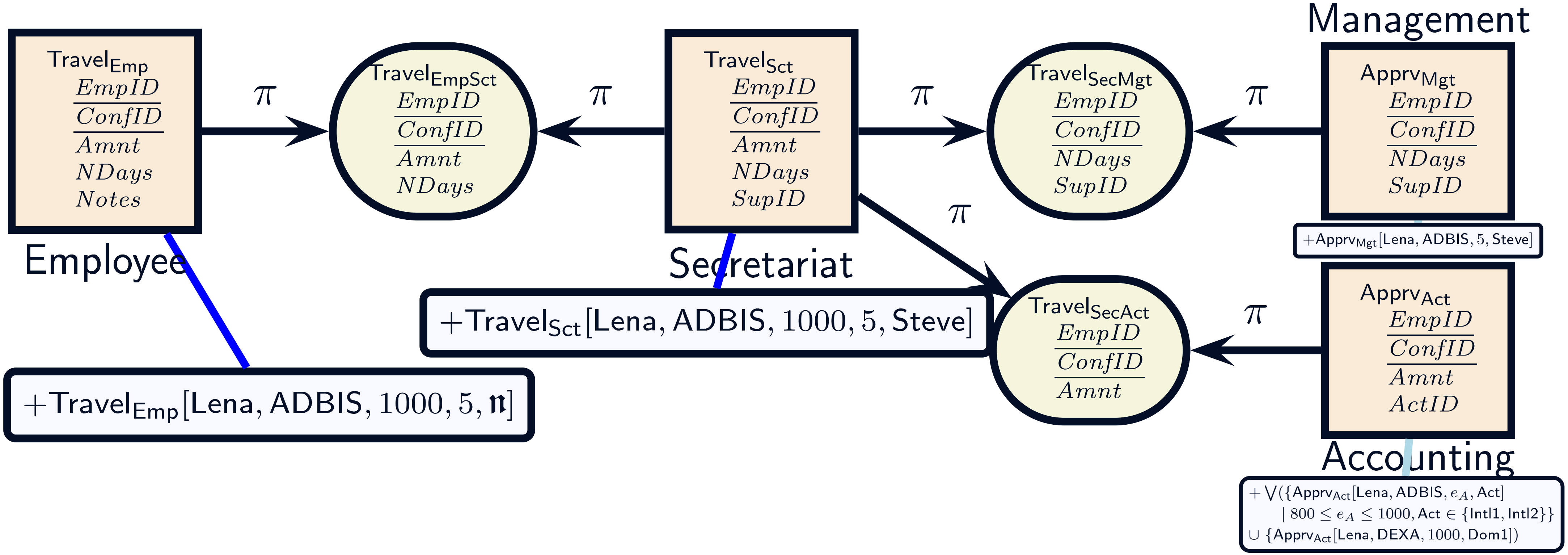
- The port-status register is cleared.
- Note that all port-status registers are now clear.

Example: Evolution of Travel Request and Authorization



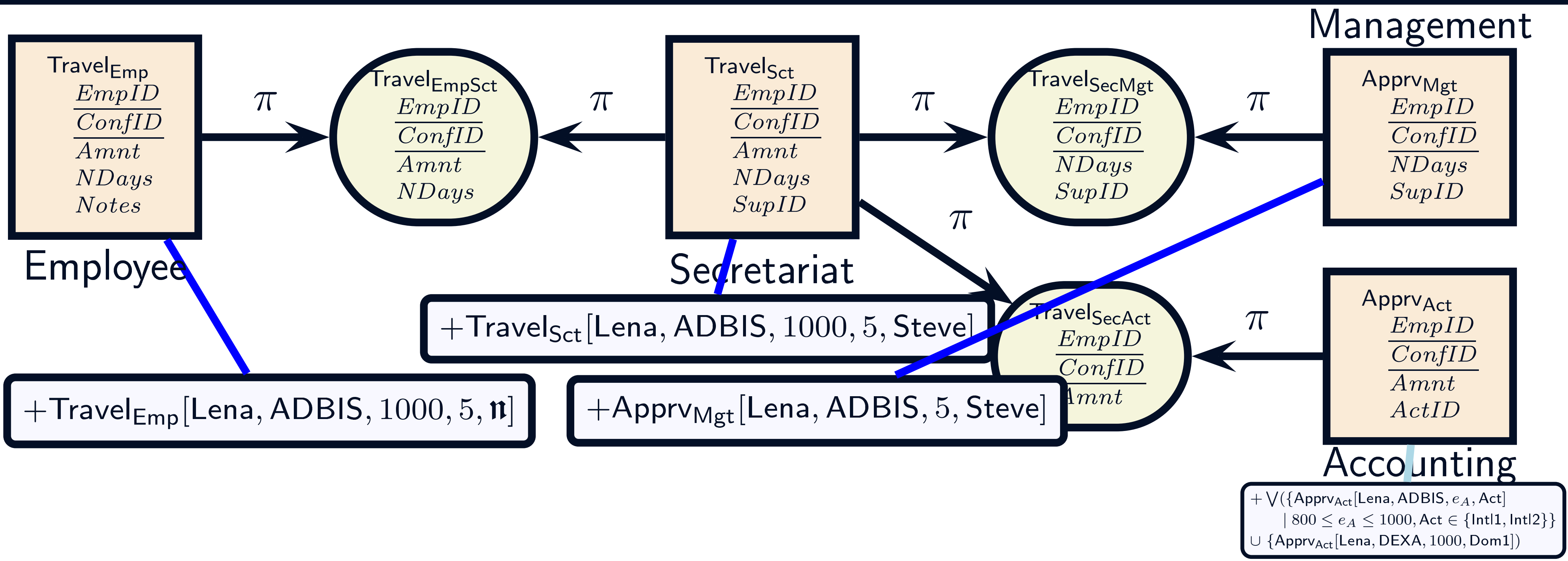
- Finally, Lena selects an update from amongst the possibilities.

Example: Evolution of Travel Request and Authorization



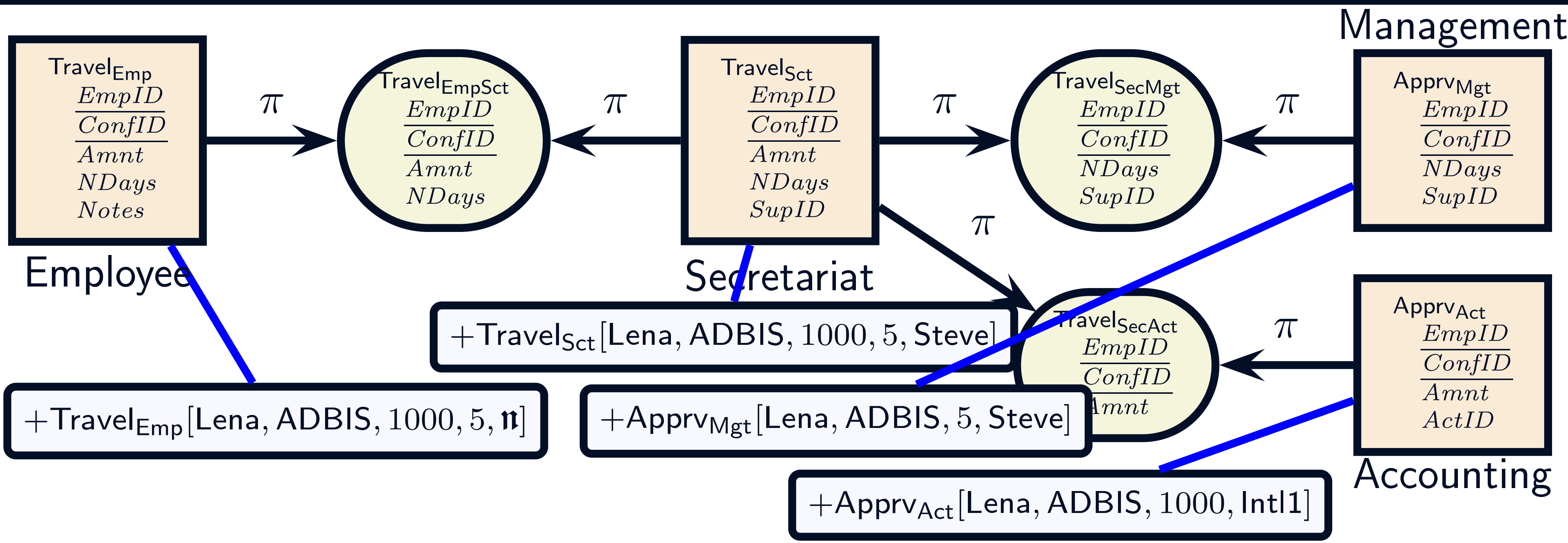
- This update is propagated to the other components for agreement.
- The messages passed through the port-status registers are not shown.

Example: Evolution of Travel Request and Authorization



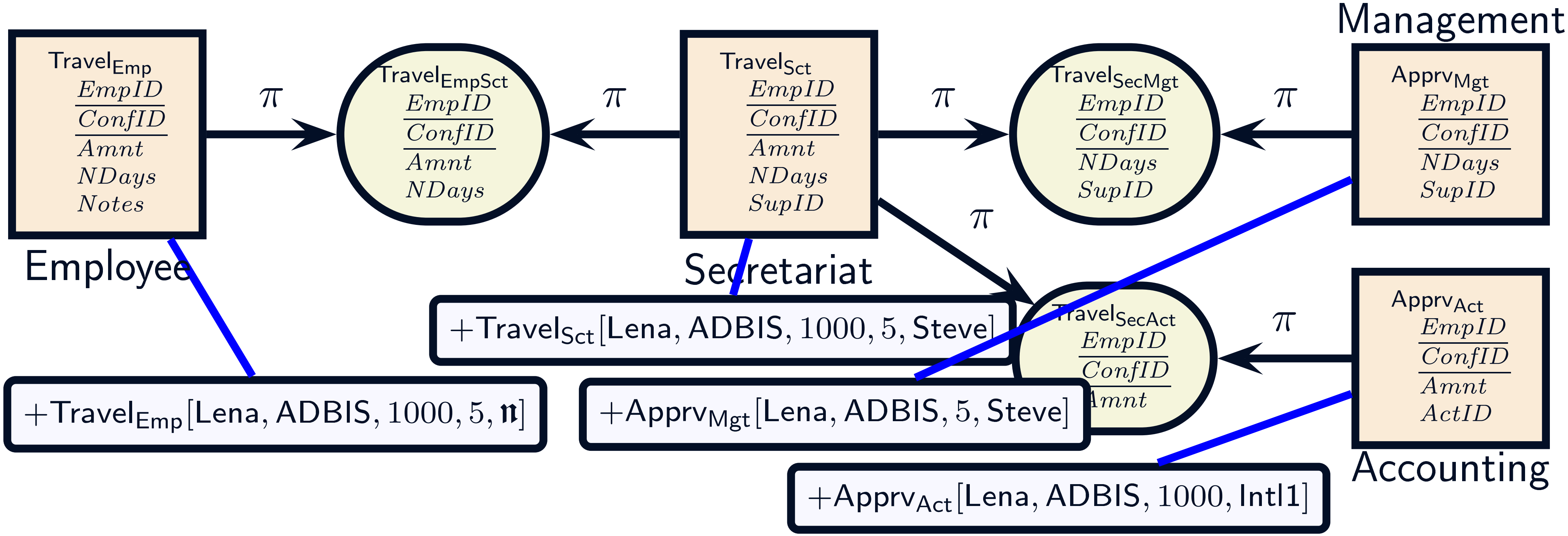
- This update is propagated to the other components for agreement.
- The messages passed through the port-status registers are not shown.

Example: Evolution of Travel Request and Authorization



- This update is propagated to the other components for agreement.
- The messages passed through the port-status registers are not shown.
- Note that a decision of which account to use is made by Accounting, but is not propagated since it is local to that component.

Example: Evolution of Travel Request and Authorization



Commit!

- Finally, these proposed updates may be committed to the database.

Current Work on Update Support

Extension of the Current work.

- The work just described is an initial “proof-of-concept” effort.

Extension of the Current work.

- The work just described is an initial "proof-of-concept" effort.
- Many issues must be addressed before it can be of practical value.

Extension of the Current work.

- The work just described is an initial "proof-of-concept" effort.
- Many issues must be addressed before it can be of practical value.
- A list of such issues will be presented next, along with a short statement of progress.

Extension of the Current work.

- The work just described is an initial "proof-of-concept" effort.
- Many issues must be addressed before it can be of practical value.
- A list of such issues will be presented next, along with a short statement of progress.

✔ Hierarchy of authority and coercion

- In the proof-of-concept model, all components have equal authority to limit a nondeterministic update proposal.
- In reality, some components/users would have more authority than others.

Example: The secretariat might have less authority than an employee applying for travel.

- This hierarchy of authority should be respected in the update negotiation process via *coercion*.

Extension of the Current Work

✔ A model of user-defined flow of control

- In the proof-of-concept model, a lifted update is immediately communicated to all neighboring components.
- Certain efficiencies can be achieved by allowing components to route update proposals to one component before the other.

Example: The secretariat might route an update request to management before sending it to accounting for a funding request.

Extension of the Current Work

✔ A model of user-defined flow of control

- In the proof-of-concept model, a lifted update is immediately communicated to all neighboring components.
- Certain efficiencies can be achieved by allowing components to route update proposals to one component before the other.

Example: The secretariat might route an update request to management before sending it to accounting for a funding request.

✔ A localized model of control and communication

- The proof-of-concept model is controlled by a global automaton.
- A localized model of control is preferable, in which all control is achieved by communication with immediate neighbors.

Extension of the Current Work

✔ A model of user-defined flow of control

- In the proof-of-concept model, a lifted update is immediately communicated to all neighboring components.
- Certain efficiencies can be achieved by allowing components to route update proposals to one component before the other.

Example: The secretariat might route an update request to management before sending it to accounting for a funding request.

✔ A localized model of control and communication

- The proof-of-concept model is controlled by a global automaton.
- A localized model of control is preferable, in which all control is achieved by communication with immediate neighbors.

✔ A model which supports simultaneous update proposals

- This is conditioned upon a suitable locking mechanism, of course.

Extension of the Current Work

✔ A bounded model of update negotiation

- In the proof-of-concept model, the update process is guaranteed to converge, but the maximum number of steps is bounded only by the number of update alternatives in all inititial liftings.
- In the current work, a much simpler model which converges in only three passes has been developed.
- It is limited to acyclic interconnections.

Extension of the Current Work

✔ A bounded model of update negotiation

- In the proof-of-concept model, the update process is guaranteed to converge, but the maximum number of steps is bounded only by the number of update alternatives in all inititial liftings.
- In the current work, a much simpler model which converges in only three passes has been developed.
- It is limited to acyclic interconnections.

✔ A model of access rights to support all of this.

Extension of the Current Work

✔ A bounded model of update negotiation

- In the proof-of-concept model, the update process is guaranteed to converge, but the maximum number of steps is bounded only by the number of update alternatives in all inititial liftings.
- In the current work, a much simpler model which converges in only three passes has been developed.
- It is limited to acyclic interconnections.

✔ A model of access rights to support all of this.

✔ An executable simulation model written in CoreASM.

Important Further Extensions

The following important issues are not yet addressed in the current work.

- ✘ Non-monotonic negotiation

Important Further Extensions

The following important issues are not yet addressed in the current work.

- ✘ Non-monotonic negotiation
- ✘ Intra-component negotiation

Important Further Extensions

The following important issues are not yet addressed in the current work.

- ✘ Non-monotonic negotiation
- ✘ Intra-component negotiation
- ✘ Relationship to workflow

Important Further Extensions

The following important issues are not yet addressed in the current work.

- ✘ Non-monotonic negotiation
- ✘ Intra-component negotiation
- ✘ Relationship to workflow
- ✘ Locking mechanism

Important Further Extensions

The following important issues are not yet addressed in the current work.

- ✘ Non-monotonic negotiation
- ✘ Intra-component negotiation
- ✘ Relationship to workflow
- ✘ Locking mechanism
- ✘ Cyclic interconnections

Key Design Goals

- The specification should be reasonably formal and complete.

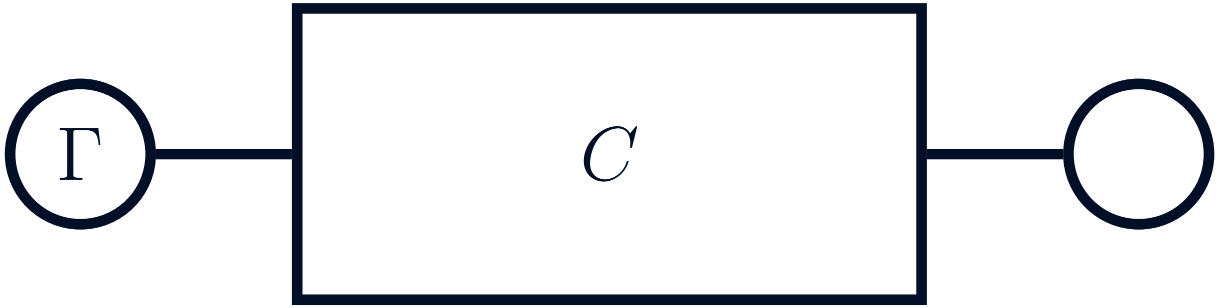
Key Design Goals

- The specification should be reasonably formal and complete.
- Key properties, such as termination of the negotiation process, should be provable.

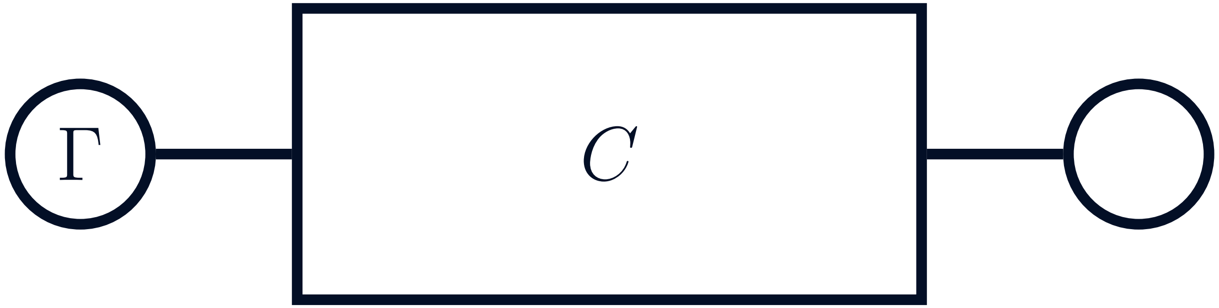
Key Design Goals

- The specification should be reasonably formal and complete.
- Key properties, such as termination of the negotiation process, should be provable.
- There should be enough detail to allow the construction of simulation models.

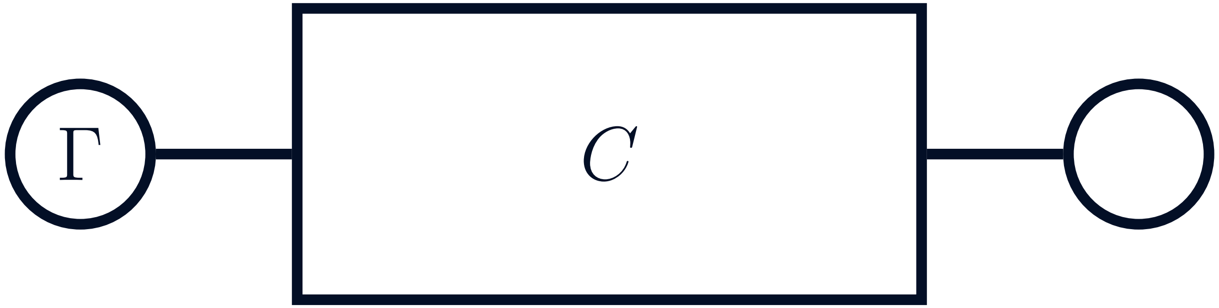
Lifting Strategies



- A key process in update support for interconnected components is the *lifting* of updates from the schema of a port Γ to the main schema of a component C.

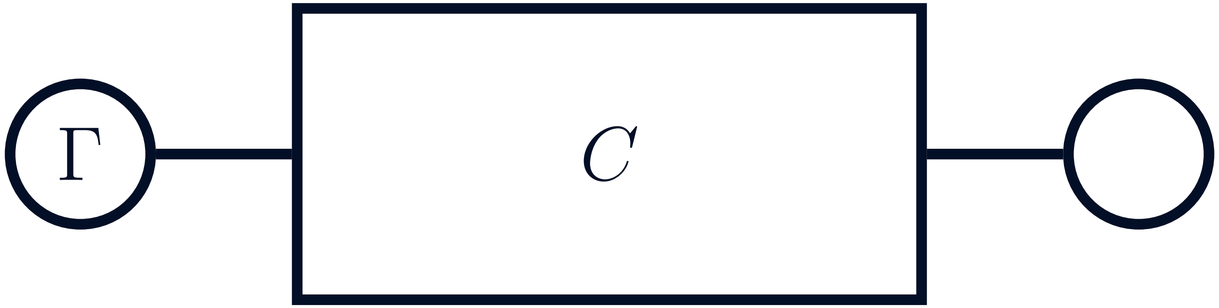


- A key process in update support for interconnected components is the *lifting* of updates from the schema of a port Γ to the main schema of a component C.
- In general, *update families* are considered.



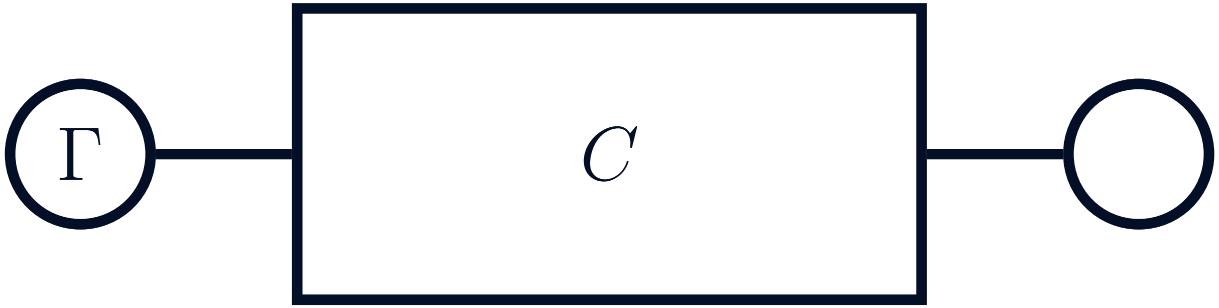
- A key process in update support for interconnected components is the *lifting* of updates from the schema of a port Γ to the main schema of a component C.
- In general, *update families* are considered.
- An *update family* is a set of updates.

Lifting Strategies



- A key process in update support for interconnected components is the *lifting* of updates from the schema of a port Γ to the main schema of a component C.
- In general, *update families* are considered.
- An *update family* is a set of updates.
- If the process succeeds, one of the members of the family will be chosen as the final update.

Lifting Strategies



- A key process in update support for interconnected components is the *lifting* of updates from the schema of a port Γ to the main schema of a component C.
- In general, *update families* are considered.
- An *update family* is a set of updates.
- If the process succeeds, one of the members of the family will be chosen as the final update.
- It is important to have some formalisms to manage this process.

Lifting Strategies and Context-Free Policies

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \rightarrow \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathsf{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

Lifting Strategies and Context-Free Policies

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathsf{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

- An *update* on $\mathbf{D}$ is a pair $(M_1, M_2) \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{LDB}(\mathbf{D})$. M_1 is called the *old state* and M_2 the *new state*.

Lifting Strategies and Context-Free Policies

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathsf{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

- An *update* on $\mathbf{D}$ is a pair $(M_1, M_2) \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{LDB}(\mathbf{D})$. M_1 is called the *old state* and M_2 the *new state*.
- Let $M_1 \in \mathsf{LDB}(\mathbf{D})$ and let (N_1, N_2) be an update on the view schema $\mathbf{V}$ with $\gamma(M_1) = N_1$. A *reflection* of (N_1, N_2) for M_1 along Γ is an update (M_1, M_2) on $\mathbf{D}$ for which $\gamma(M_2) = N_2$.

$$\begin{tikzcd} M_1 \arrow[r] \arrow[d, "\gamma"'] & M_2 \arrow[d, "\gamma"] \\ N_1 \arrow[r] & N_2 \end{tikzcd}$$

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathsf{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

- An *update* on $\mathbf{D}$ is a pair $(M_1, M_2) \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{LDB}(\mathbf{D})$. M_1 is called the *old state* and M_2 the *new state*.

- Let $M_1 \in \mathsf{LDB}(\mathbf{D})$ and let (N_1, N_2) be an update on the view schema $\mathbf{V}$ with $\gamma(M_1) = N_1$. A *reflection* of (N_1, N_2) for M_1 along Γ is an update (M_1, M_2) on $\mathbf{D}$ for which $\gamma(M_2) = N_2$.

$$\begin{tikzcd} M_1 \arrow[r] \arrow[d, "\gamma"'] & M_2 \arrow[d, "\gamma"] \\ N_1 \arrow[r] & N_2 \end{tikzcd}$$

- For $M_1 \in \mathsf{LDB}(\mathbf{D})$, an *update family* for M_1 is a nonempty set of updates on $\mathbf{D}$, each with old state M_1.

Lifting Strategies and Context-Free Policies

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathrm{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

- An *update* on $\mathbf{D}$ is a pair $(M_1, M_2) \in \mathrm{LDB}(\mathbf{D}) \times \mathrm{LDB}(\mathbf{D})$. M_1 is called the *old state* and M_2 the *new state*.

- Let $M_1 \in \mathrm{LDB}(\mathbf{D})$ and let (N_1, N_2) be an update on the view schema $\mathbf{V}$ with $\gamma(M_1) = N_1$. A *reflection* of (N_1, N_2) for M_1 along Γ is an update (M_1, M_2) on $\mathbf{D}$ for which $\gamma(M_2) = N_2$.

$$\begin{tikzcd} M_1 \arrow[r] \arrow[d, "\gamma"'] & M_2 \arrow[d, "\gamma"] \\ N_1 \arrow[r] & N_2 \end{tikzcd}$$

- For $M_1 \in \mathrm{LDB}(\mathbf{D})$, an *update family* for M_1 is a nonempty set of updates on $\mathbf{D}$, each with old state M_1.

- For $N_1 \in \mathrm{LDB}(\mathbf{V})$, (N_1, N_2) an update on $\mathbf{V}$, and $M_1 \in \mathrm{LDB}(\mathbf{D})$ with $\gamma(M_1) = N_1$, a *lifting* of (N_1, N_2) to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $N_2 = \gamma(M_2)$.

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathsf{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

- For $N_1 \in \mathsf{LDB}(\mathbf{V})$, (N_1, N_2) an update on $\mathbf{V}$, and $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $\gamma(M_1) = N_1$, a *lifting* of (N_1, N_2) to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $N_2 = \gamma(M_2)$.

- For $N_1 \in \mathsf{LDB}(\mathbf{V})$, $\mathbf{u}'$ an update family for N_1, and $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $\gamma(M_1) = N_1$, a *lifting* of $\mathbf{u}$ to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $(N_1, \gamma(M_2)) \in \mathbf{u}$.

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: $\mathsf{LDB}(\mathbf{D})$ = *legal databases* of $\mathbf{D}$.

- For $N_1 \in \mathsf{LDB}(\mathbf{V})$, (N_1, N_2) an update on $\mathbf{V}$, and $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $\gamma(M_1) = N_1$, a *lifting* of (N_1, N_2) to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $N_2 = \gamma(M_2)$.

- For $N_1 \in \mathsf{LDB}(\mathbf{V})$, $\mathbf{u}'$ an update family for N_1, and $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $\gamma(M_1) = N_1$, a *lifting* of $\mathbf{u}$ to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $(N_1, \gamma(M_2)) \in \mathbf{u}$.

- A *lifting strategy* for Γ is a function λ which assigns to each $M_1 \in \mathsf{LDB}(\mathbf{D})$ and update family $\mathbf{u}$ for $\gamma(M_1)$ a lifting $\lambda\langle M_1, \mathbf{u}\rangle$ of $\mathbf{u}$ to $\mathbf{D}$ for M_1.

Lifting Strategies and Context-Free Policies

Context: $\mathbf{D}$ = database schema; $\Gamma = (\mathbf{V}, \gamma : \mathbf{D} \to \mathbf{V})$ a view of $\mathbf{D}$.

Notation: LDB($\mathbf{D}$) = *legal databases* of $\mathbf{D}$.

- For $N_1 \in$ LDB($\mathbf{V}$), (N_1, N_2) an update on $\mathbf{V}$, and $M_1 \in$ LDB($\mathbf{D}$) with $\gamma(M_1) = N_1$, a *lifting* of (N_1, N_2) to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $N_2 = \gamma(M_2)$.

- For $N_1 \in$ LDB($\mathbf{V}$), $\mathbf{u}'$ an update family for N_1, and $M_1 \in$ LDB($\mathbf{D}$) with $\gamma(M_1) = N_1$, a *lifting* of $\mathbf{u}$ to $\mathbf{D}$ for M_1 is an update family $\mathbf{u}$ for M_1 such that for every $(M_1, M_2) \in \mathbf{u}$, $(N_1, \gamma(M_2)) \in \mathbf{u}$.

- A *lifting strategy* for Γ is a function λ which assigns to each $M_1 \in$ LDB($\mathbf{D}$) and update family $\mathbf{u}$ for $\gamma(M_1)$ a lifting $\lambda\langle M_1, \mathbf{u}\rangle$ of $\mathbf{u}$ to $\mathbf{D}$ for M_1.

- The lifting strategy λ for Γ is *context free* if for every $M_1 \in$ LDB($\mathbf{D}$) and every update family $\mathbf{u}'$ for $\gamma(M_1)$,

$$\lambda\langle M_1, \mathbf{u}\rangle = \bigcup\{\lambda\langle M_1, \{u\}\rangle \mid u \in \mathbf{u}\}$$

Minimality of Liftings

- It is important that lifted updates be *minimal*, in the sense that no changes beyond those necessary are made.

Minimality of Liftings

- It is important that lifted updates be *minimal*, in the sense that no changes beyond those necessary are made.
- The idea is to avoid “pork-barrel” additions which have nothing to do with the original request.

Minimality of Liftings

- It is important that lifted updates be *minimal*, in the sense that no changes beyond those necessary are made.
- The idea is to avoid "pork-barrel" additions which have nothing to do with the original request.
- For simple insertions and deletion, minimality defined by set inclusion is usually adequate.

Minimality of Liftings

- It is important that lifted updates be *minimal*, in the sense that no changes beyond those necessary are made.
- The idea is to avoid "pork-barrel" additions which have nothing to do with the original request.
- For simple insertions and deletion, minimality defined by set inclusion is usually adequate.
- For general updates, a notion of minimality based upon *information content* is appropriate [Hegner FoIKS08] [Hegner SDKB 08].

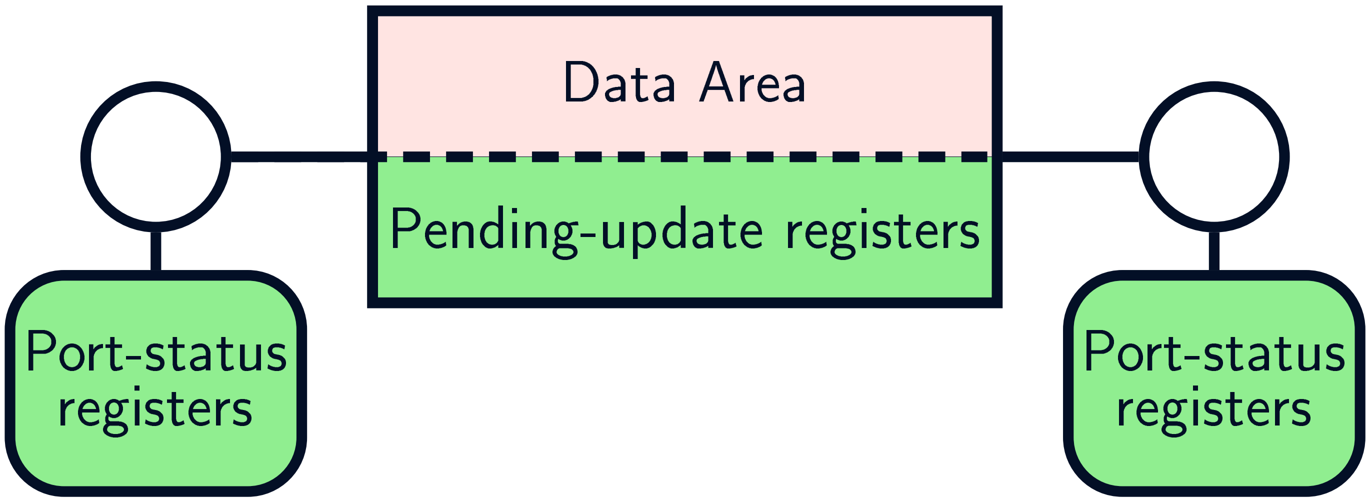
Minimality of Liftings

- It is important that lifted updates be *minimal*, in the sense that no changes beyond those necessary are made.
- The idea is to avoid "pork-barrel" additions which have nothing to do with the original request.
- For simple insertions and deletion, minimality defined by set inclusion is usually adequate.
- For general updates, a notion of minimality based upon *information content* is appropriate [Hegner FoIKS08] [Hegner SDKB 08].
- Minimality will not be discussed further in this talk, but will be assumed.

Minimality of Liftings

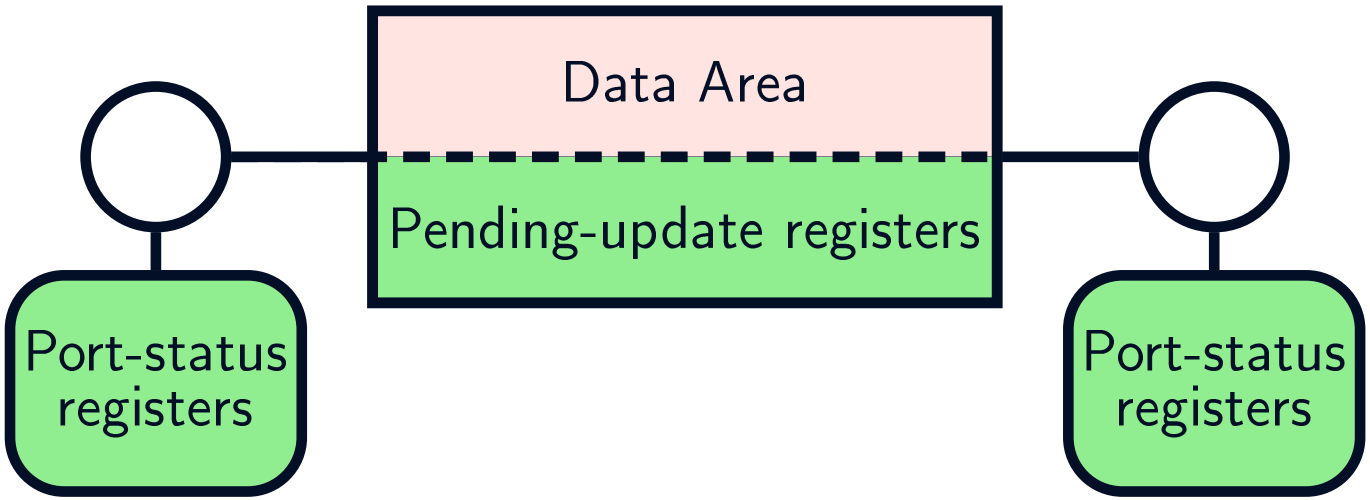
- It is important that lifted updates be *minimal*, in the sense that no changes beyond those necessary are made.
- The idea is to avoid "pork-barrel" additions which have nothing to do with the original request.
- For simple insertions and deletion, minimality defined by set inclusion is usually adequate.
- For general updates, a notion of minimality based upon *information content* is appropriate [Hegner FoIKS08] [Hegner SDKB 08].
- Minimality will not be discussed further in this talk, but will be assumed.
- Another important concept is the *minmax lifting*, which consists of the set of all minimal liftings for an update family on the view.

The Architecture of a Component



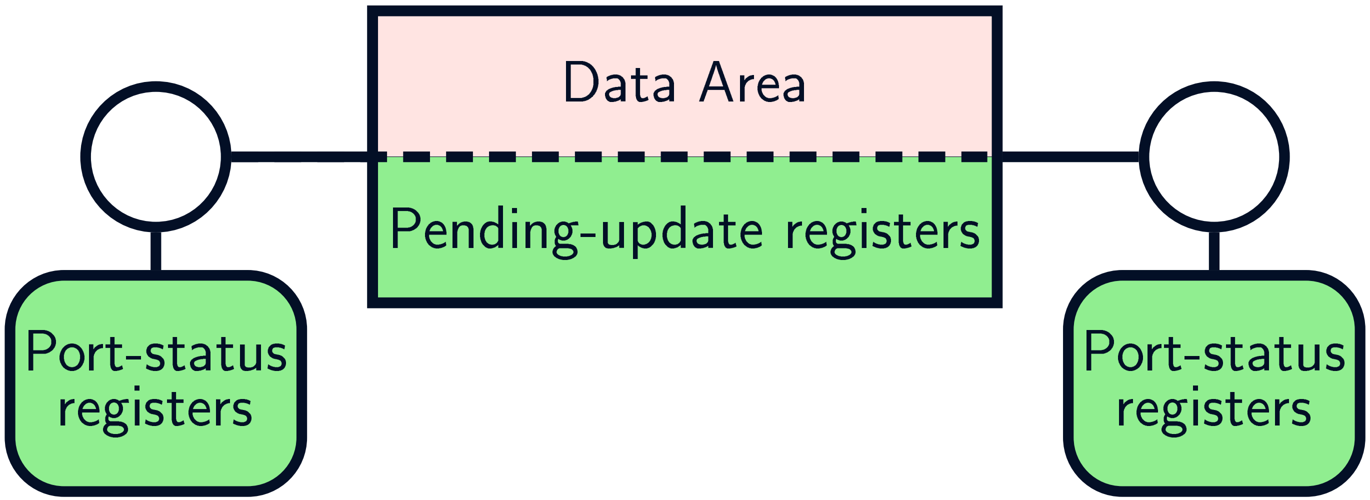
- Each update request is assigned a *Transaction ID* which is attached to all inter-component communications about that request.

The Architecture of a Component



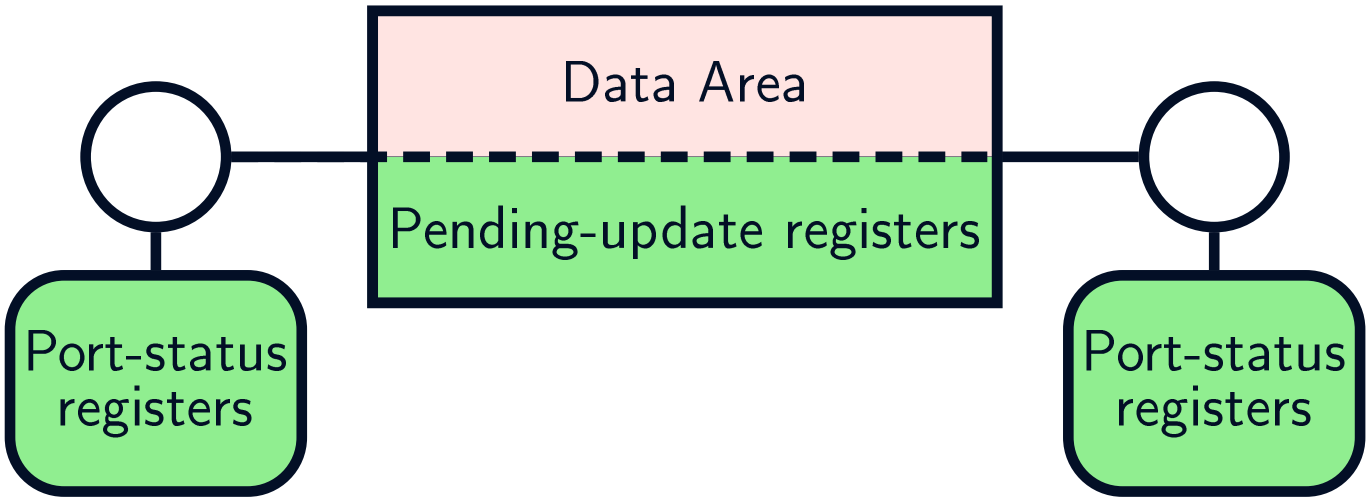
- Each update request is assigned a *Transaction ID* which is attached to all inter-component communications about that request.
- In a system which supports concurrency, the *pending-update registers* and the *port-status registers* both occur on a per-transaction-ID basis.

The Architecture of a Component

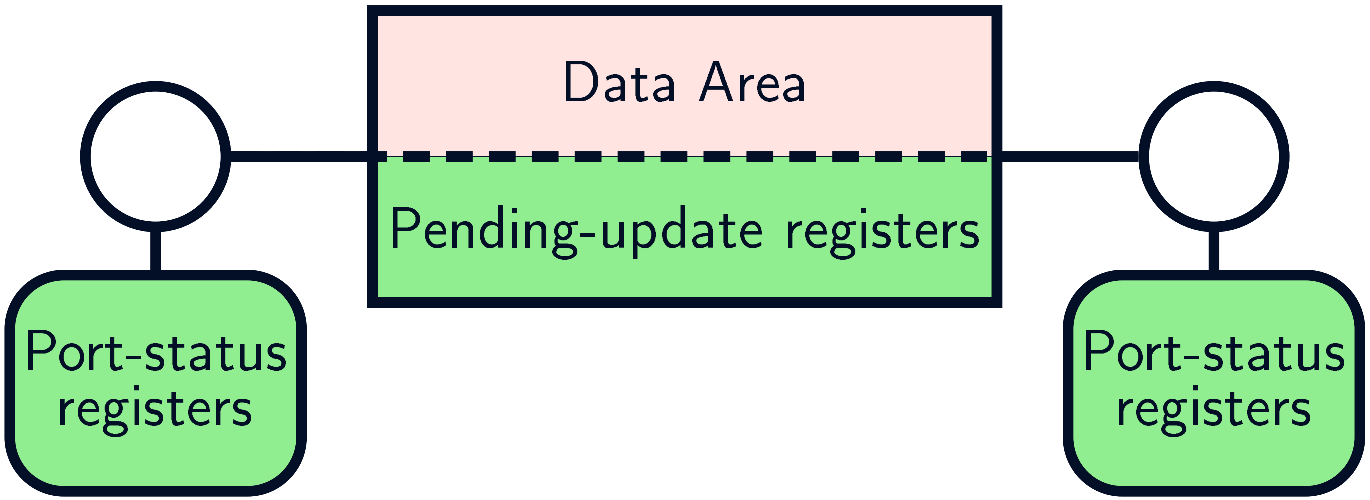


- Each update request is assigned a *Transaction ID* which is attached to all inter-component communications about that request.
- In a system which supports concurrency, the *pending-update registers* and the *port-status registers* both occur on a per-transaction-ID basis.
- The *pending-update register* for a given transaction records the current (possibly nondeterministic) update which is proposed on the given component.

The Architecture of a Component

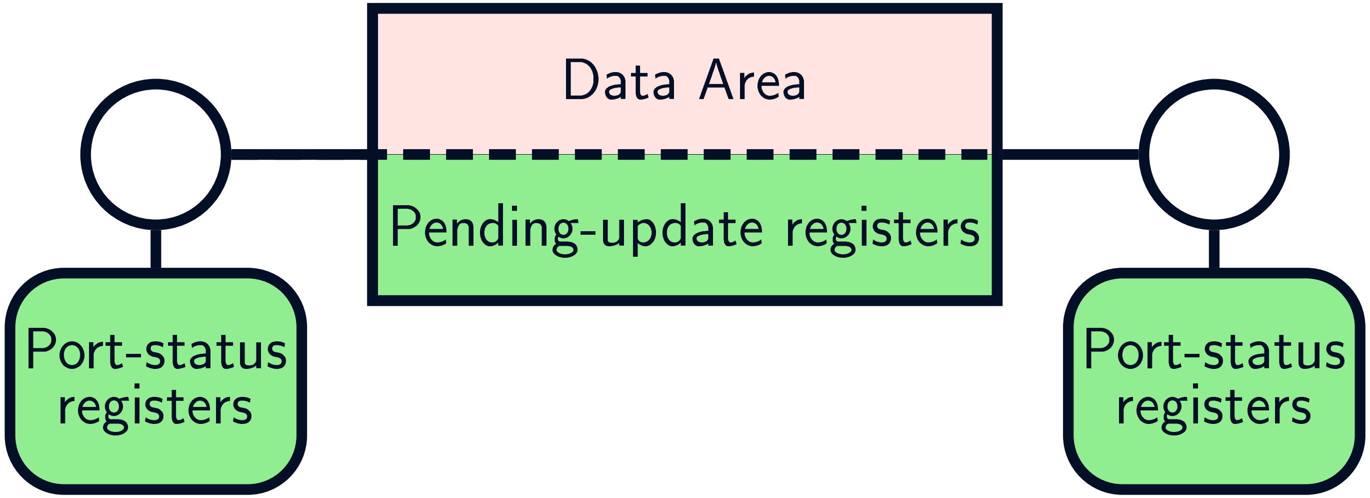


- Each update request is assigned a *Transaction ID* which is attached to all inter-component communications about that request.
- In a system which supports concurrency, the *pending-update registers* and the *port-status registers* both occur on a per-transaction-ID basis.
- The *pending-update register* for a given transaction records the current (possibly nondeterministic) update which is proposed on the given component.
- The *port-status registers* contain:
  - The current (possibly nondeterministic) update to be applied at that port.
  - State and control information necessary for support of the update process.



- Overview of the fields of a pending-update register:

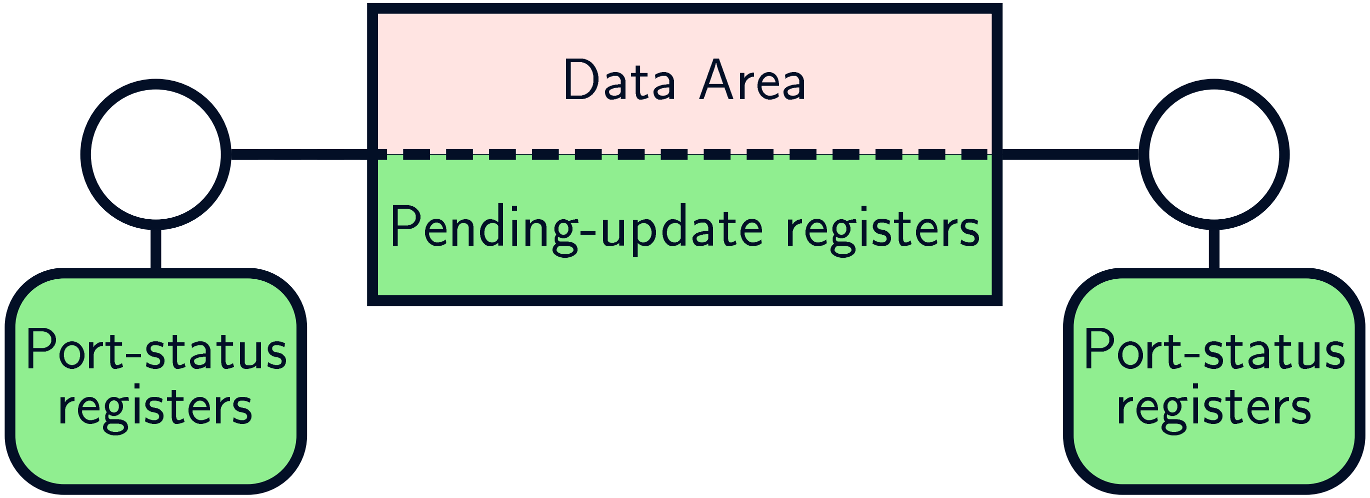
The Architecture of a Component



- Overview of the fields of a pending-update register:

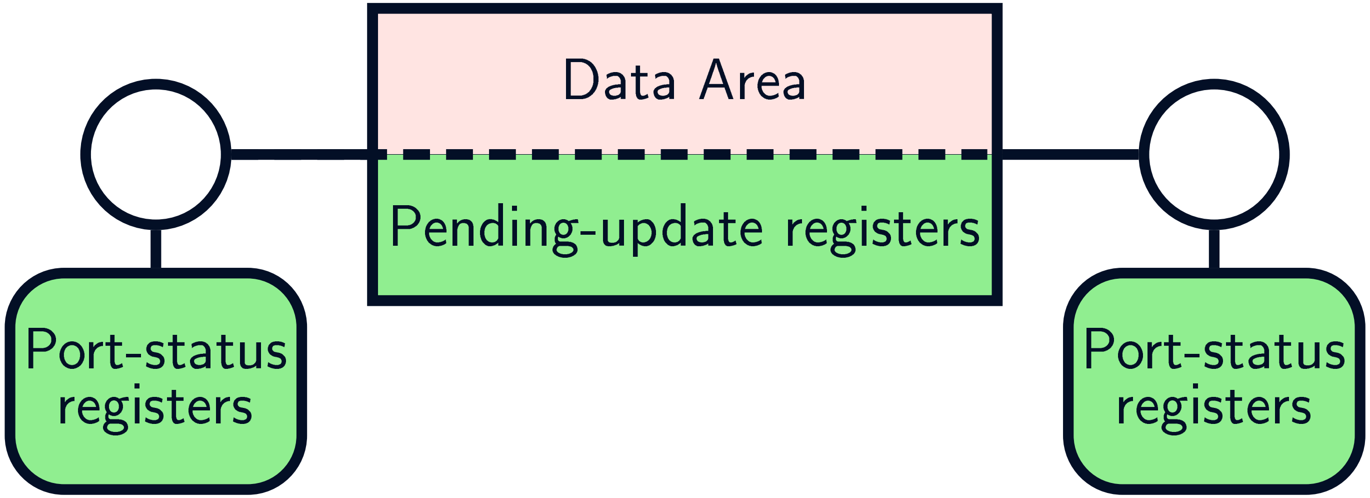
`Transaction_ID:`	ID_type
`Transaction_rank:`	Authority_class
`Pending database state:`	update family
`Current:`	Boolean
`Lock region:`	View

The Architecture of a Component



- Overview of the fields of a port-status register:

The Architecture of a Component

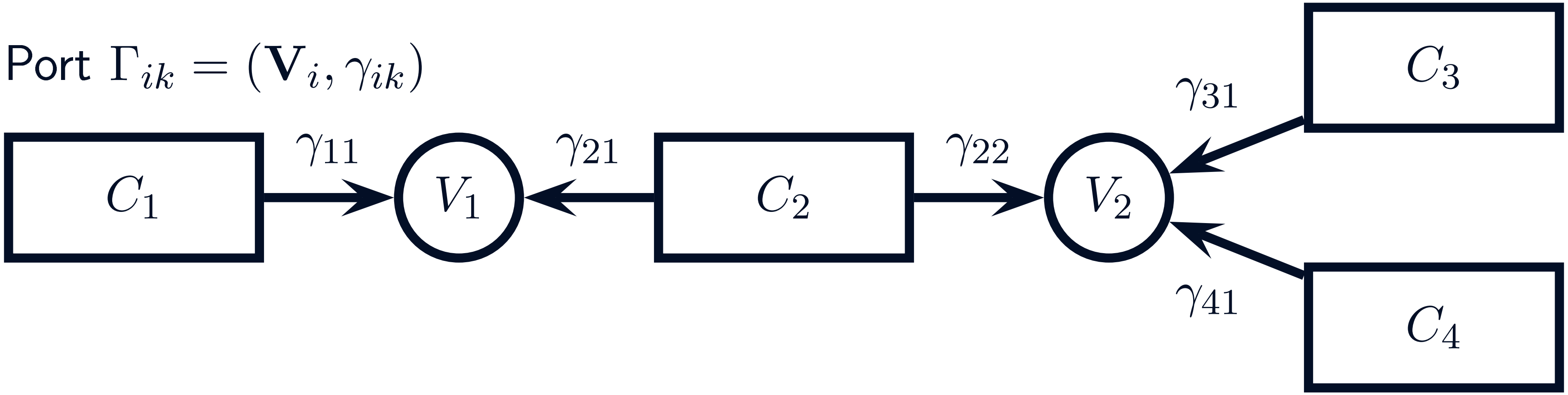


- Overview of the fields of a port-status register:

`Transaction_ID:`	ID_type
`Transaction_rank:`	Authority_class
`Pending port state:`	update family
`Type:`	{Inner, Outer}
`Status:`	{NULL, Initial, Revised} (Inner)
	{NULL, Preliminary, Final} (Outer)
Writers:	C_1: Boolean
	$\vdots$ (Neighboring components)
	C_k: Boolean

Inner Ports, Outer Ports, and Extreme Components

Port $\Gamma_{ik} = (\mathbf{V}_i, \gamma_{ik})$

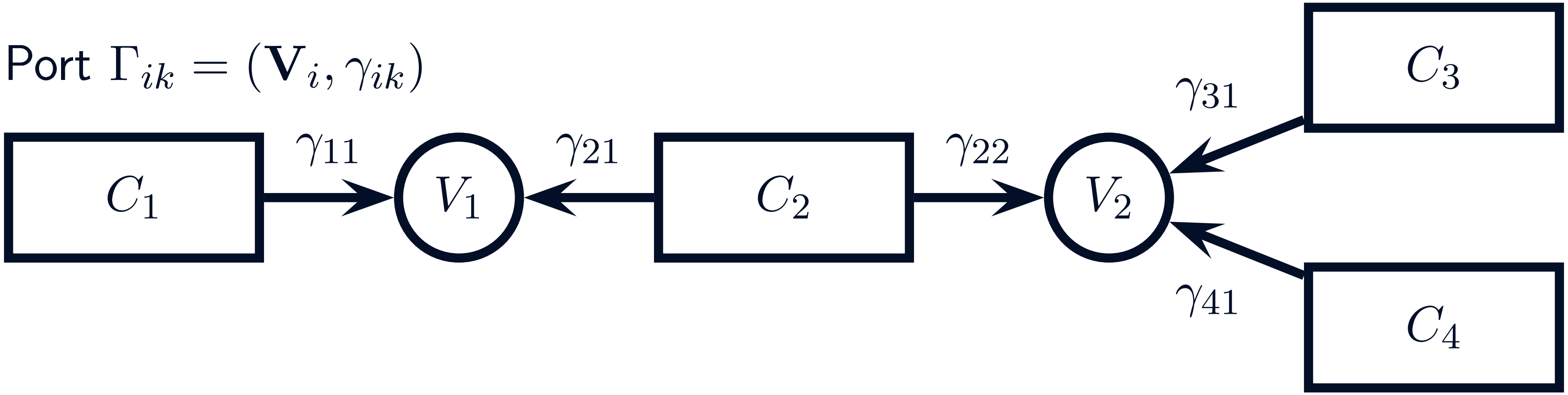


Context: Acyclic interconnection family J, C_i and C_j components.

Think: C_i is the component which initiates the update request.
C_j is any other component.

Inner Ports, Outer Ports, and Extreme Components

Port $\Gamma_{ik} = (\mathbf{V}_i, \gamma_{ik})$



Context: Acyclic interconnection family J, C_i and C_j components.

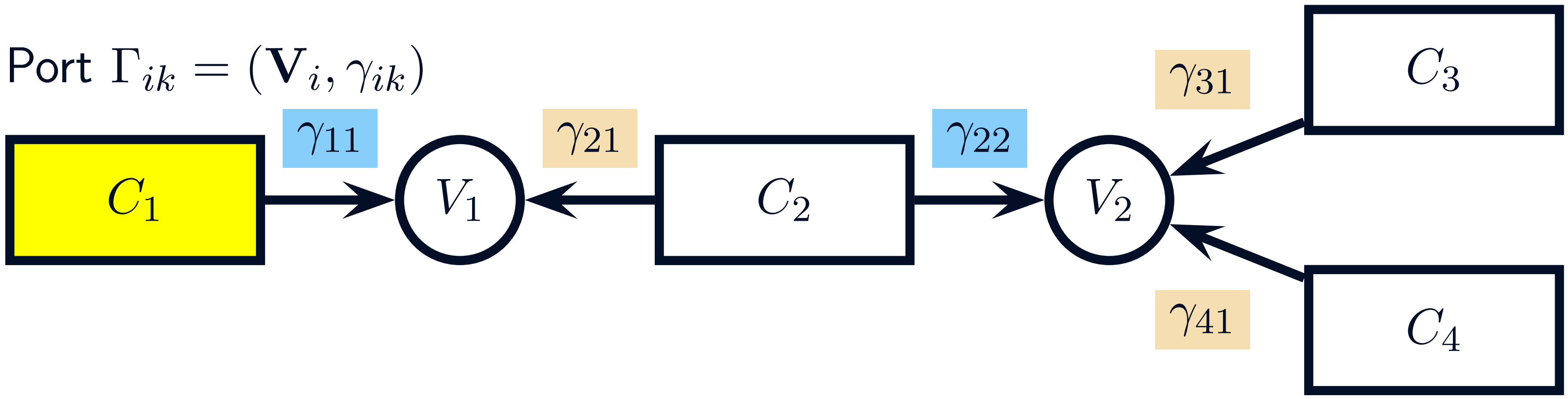
Think: C_i is the component which initiates the update request.
C_j is any other component.

Inner: The port Γ_{jk} of C_j is *inner* with respect to $\langle J, C_i \rangle$ if it lies on the path from C_i to C_j.

Outer: Otherwise it is *outer*.

- All ports of the initiating component are also outer.

Inner Ports, Outer Ports, and Extreme Components



Context: Acyclic interconnection family J, C_i and C_j components.

Think: C_i is the component which initiates the update request.
C_j is any other component.

Inner: The port Γ_{jk} of C_j is *inner* with respect to $\langle J, C_i\rangle$ if it lies on the path from C_i to C_j.

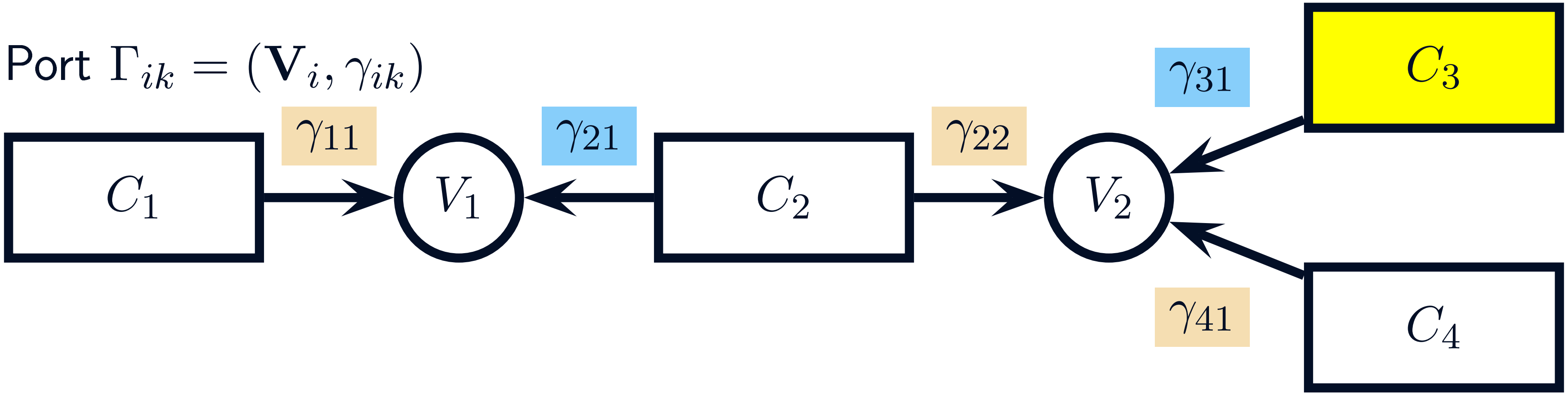
Outer: Otherwise it is *outer*.

- All ports of the initiating component are also outer.

Examples: Initial component in yellow.

Inner Ports, Outer Ports, and Extreme Components

Port $\Gamma_{ik} = (\mathbf{V}_i, \gamma_{ik})$



Context: Acyclic interconnection family J, C_i and C_j components.

Think: C_i is the component which initiates the update request.
C_j is any other component.

Inner: The port Γ_{jk} of C_j is *inner* with respect to $\langle J, C_i \rangle$ if it lies on the path from C_i to C_j.

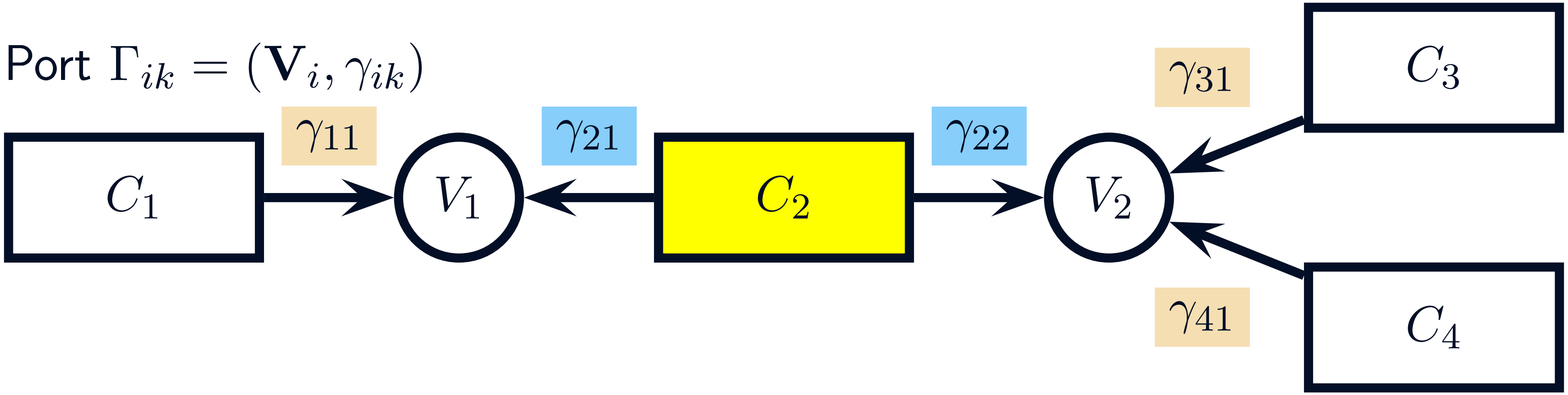
Outer: Otherwise it is *outer*.

- All ports of the initiating component are also outer.

Examples: Initial component in yellow.

Inner Ports, Outer Ports, and Extreme Components

Port $\Gamma_{ik} = (\mathbf{V}_i, \gamma_{ik})$



Context: Acyclic interconnection family J, C_i and C_j components.

Think: C_i is the component which initiates the update request.
C_j is any other component.

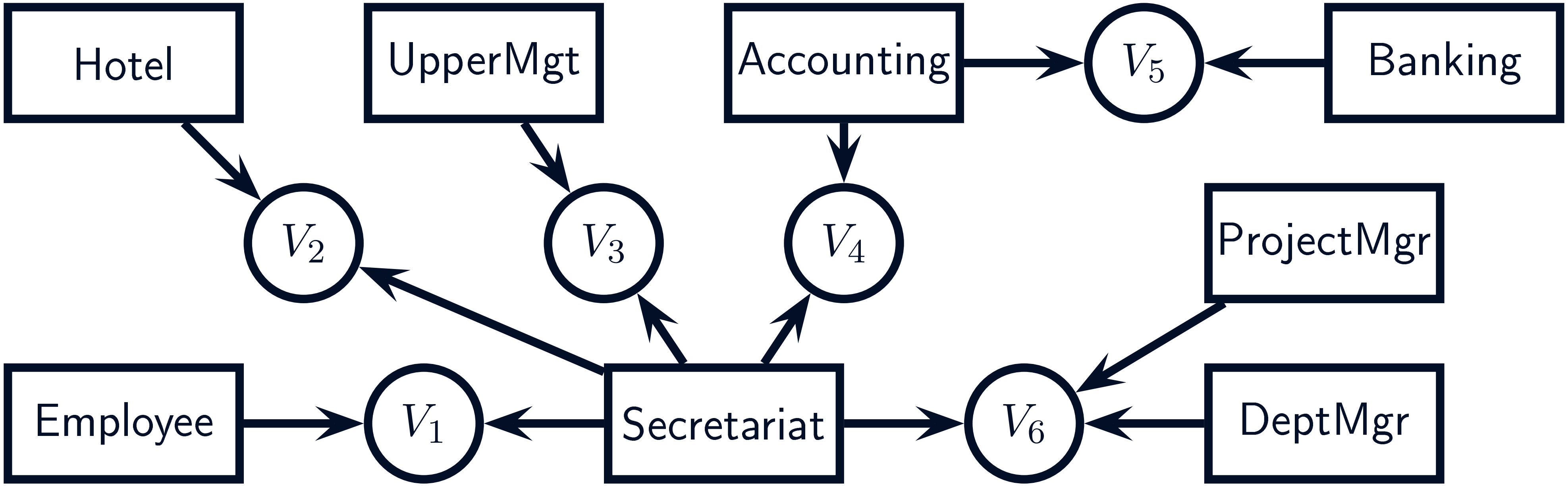
Inner: The port Γ_{jk} of C_j is *inner* with respect to $\langle J, C_i \rangle$ if it lies on the path from C_i to C_j.

Outer: Otherwise it is *outer*.

- All ports of the initiating component are also outer.

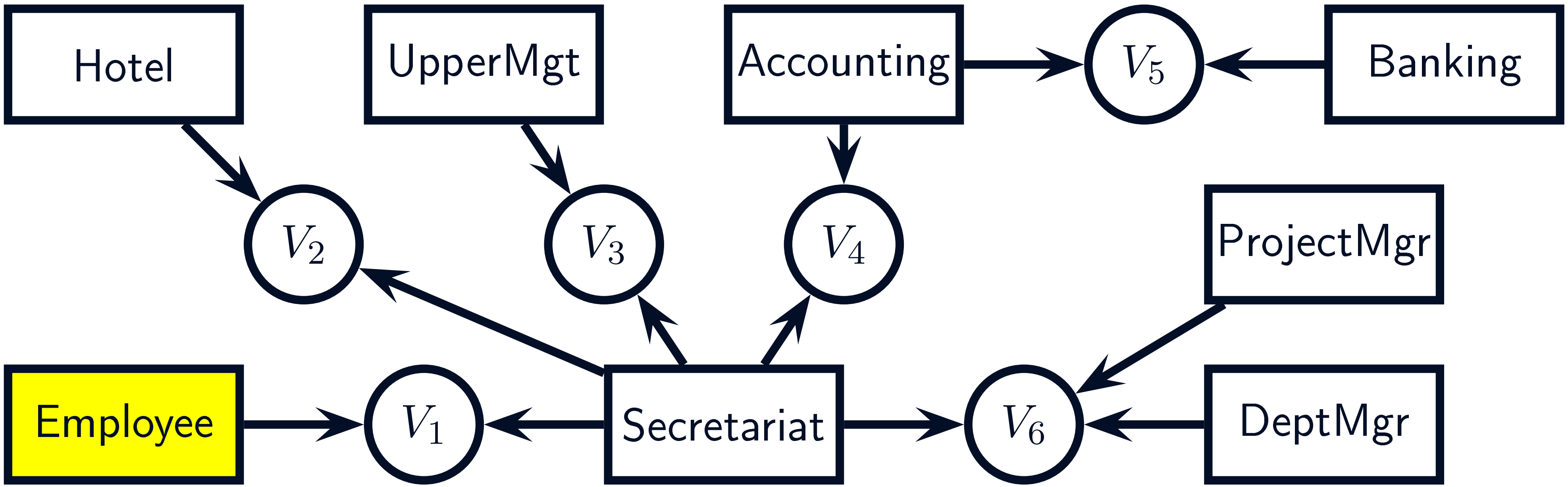
Examples: Initial component in yellow.

A More Complex Example to Illustrate All Aspects

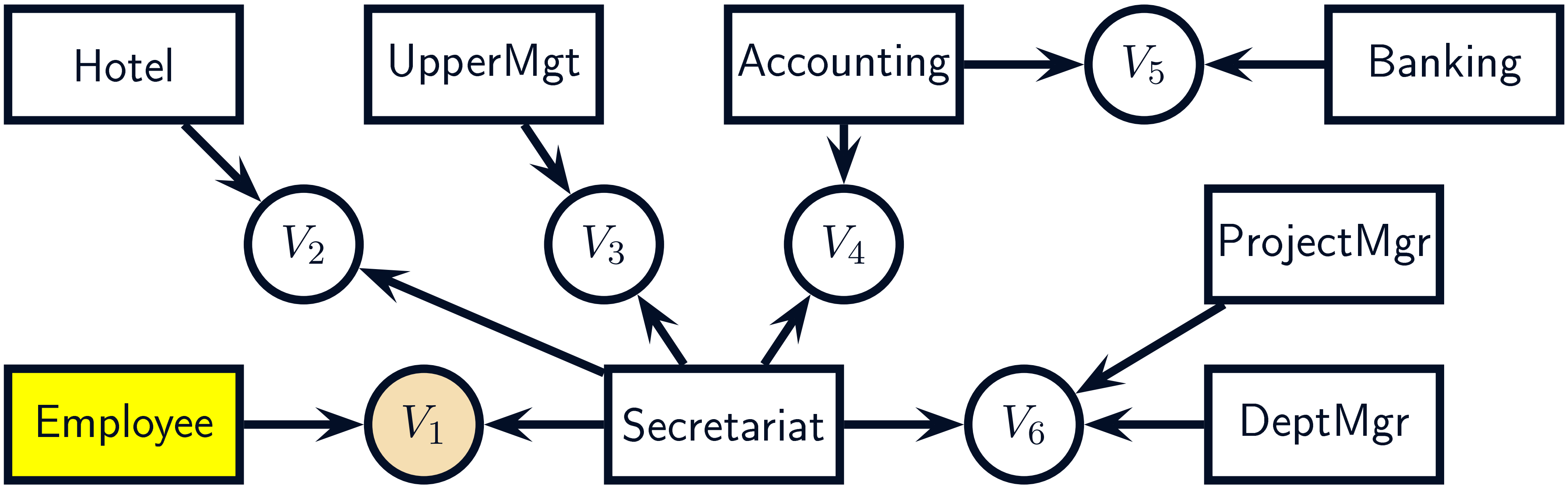


- The *Hotel* and *Banking* *components* are autonomous.
- They are automated suppliers of information; there are no human decisions involved.

The Three-Stage Update-Negotiation Process

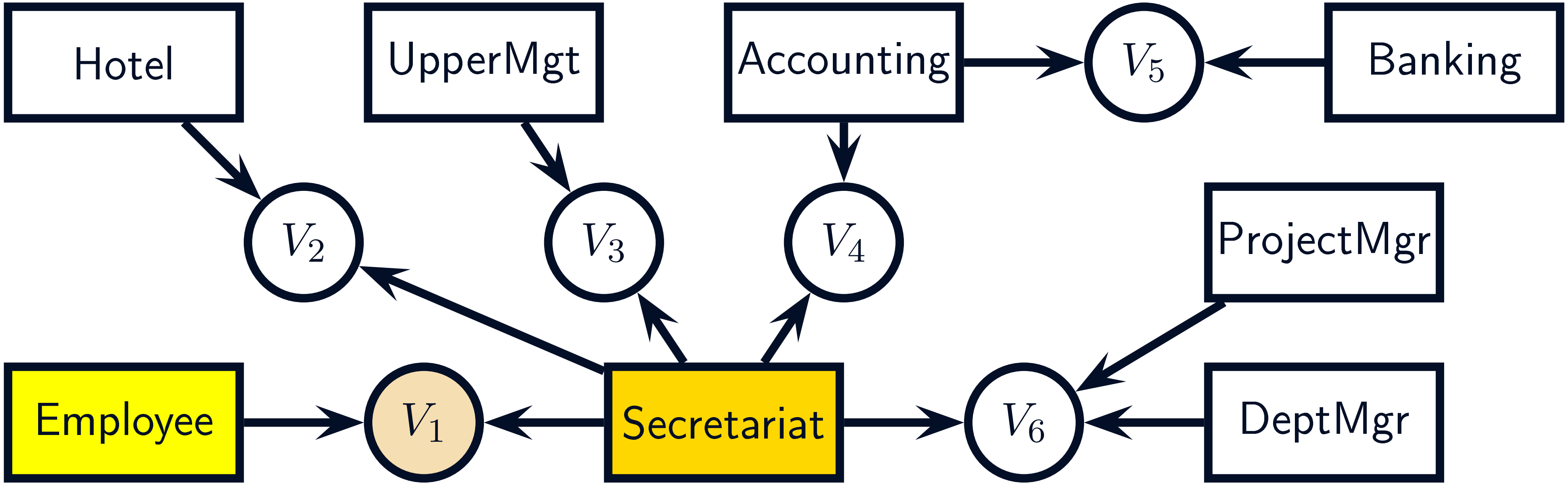


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.



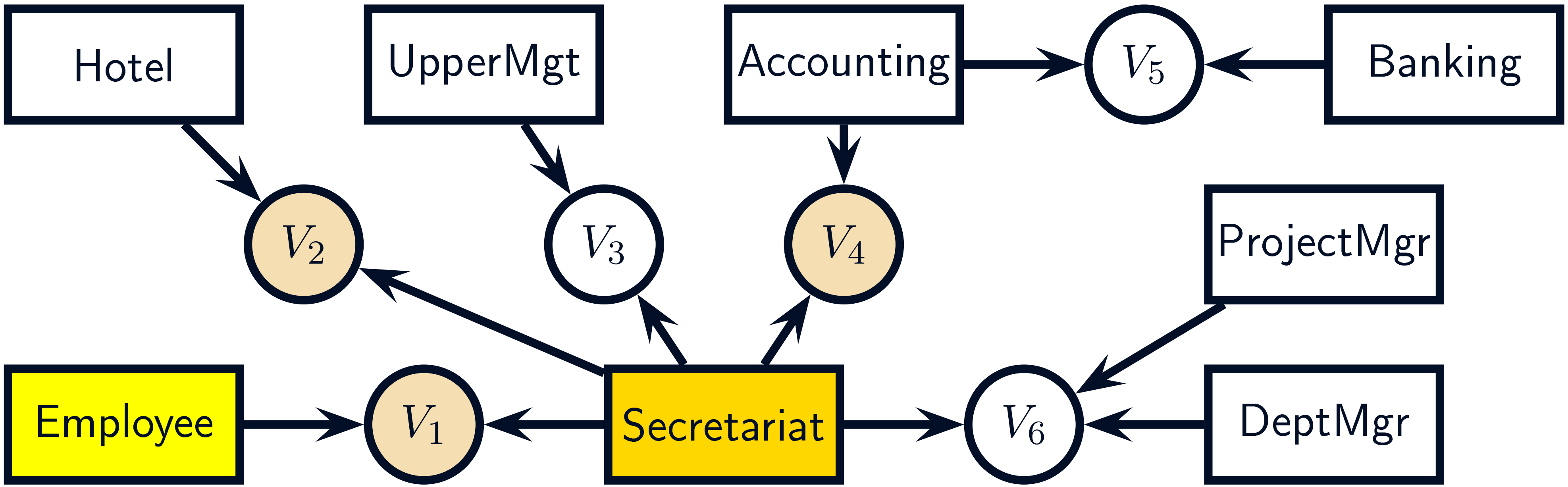
Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

The Three-Stage Update-Negotiation Process



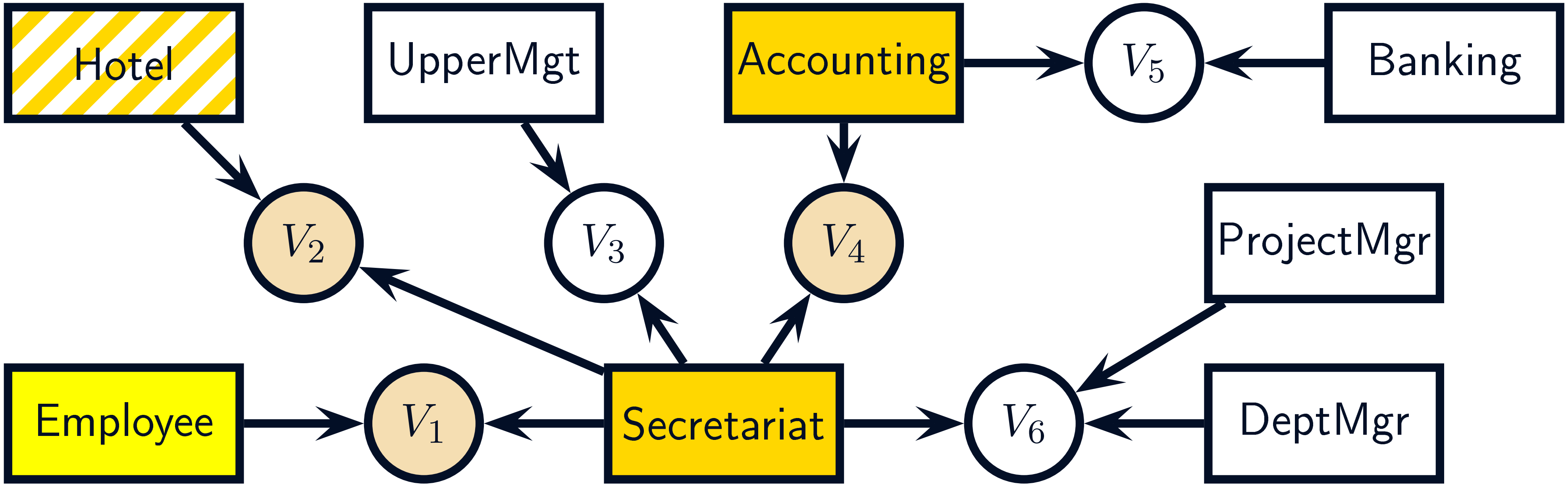
Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

The Three-Stage Update-Negotiation Process



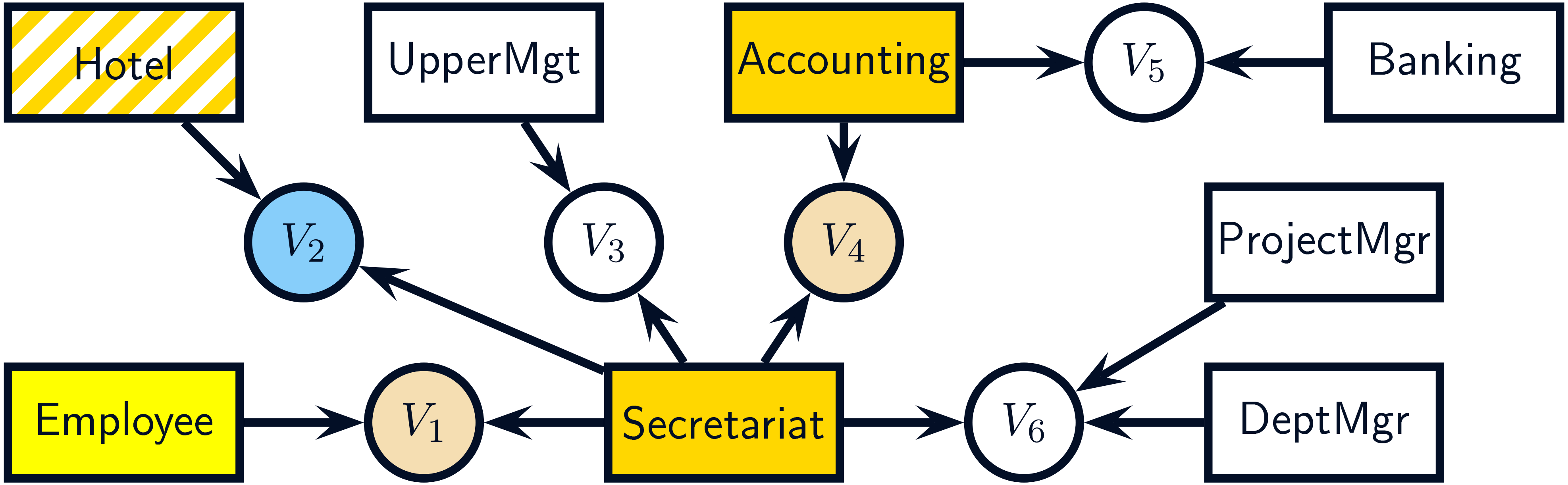
Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

The Three-Stage Update-Negotiation Process



Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

The Three-Stage Update-Negotiation Process

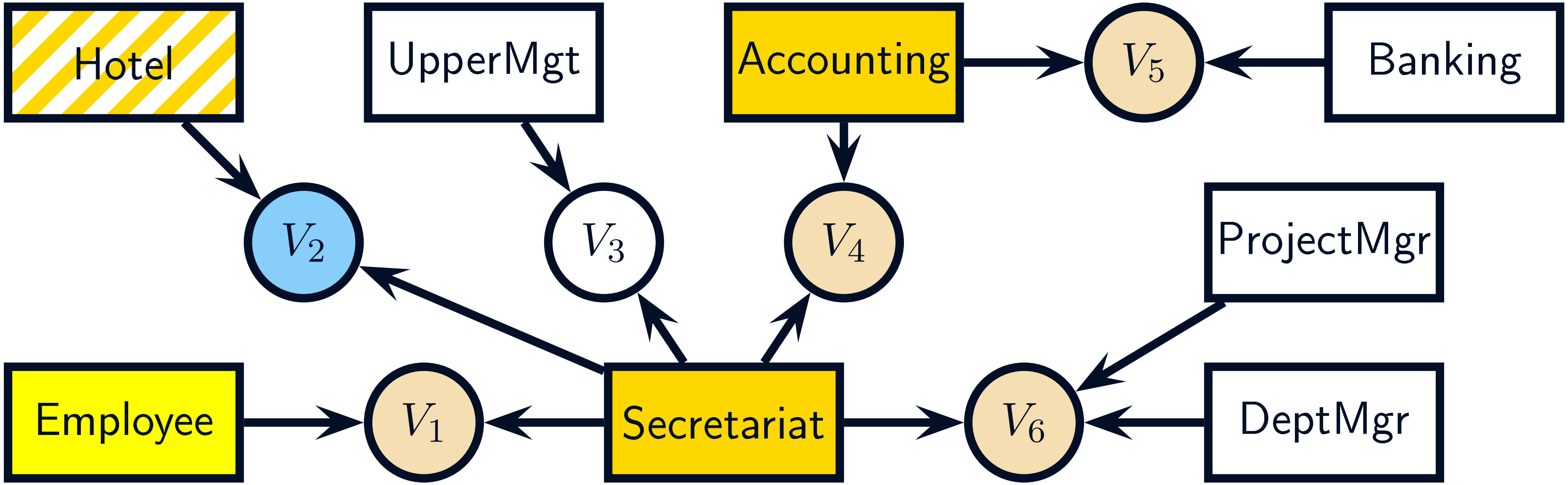


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

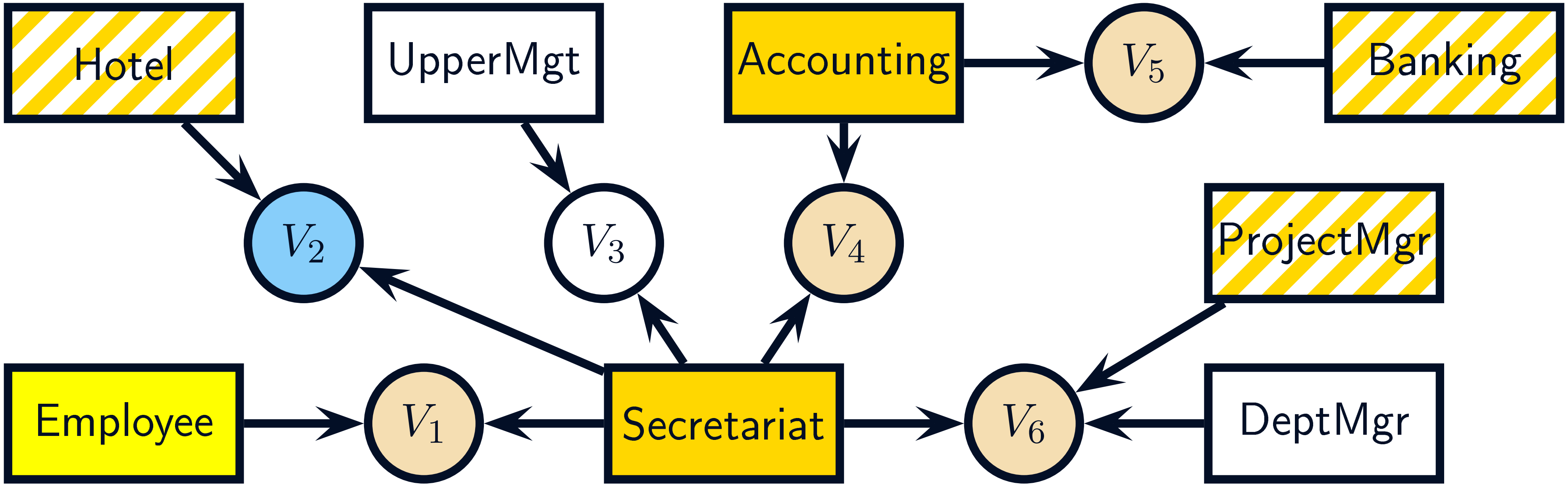


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

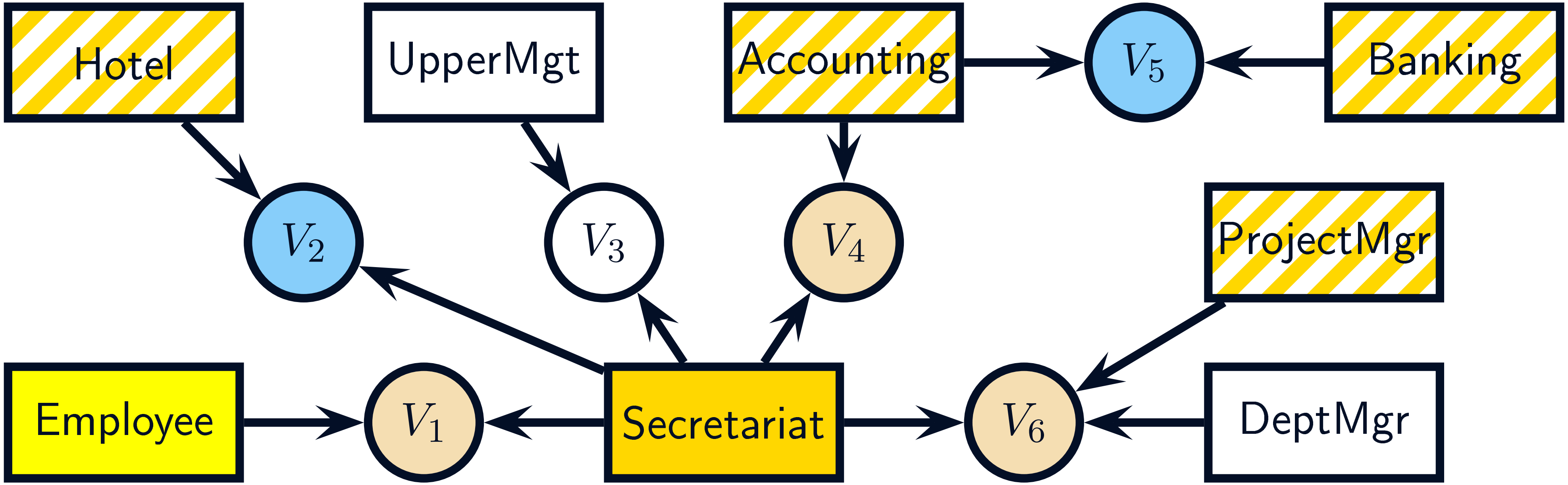


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

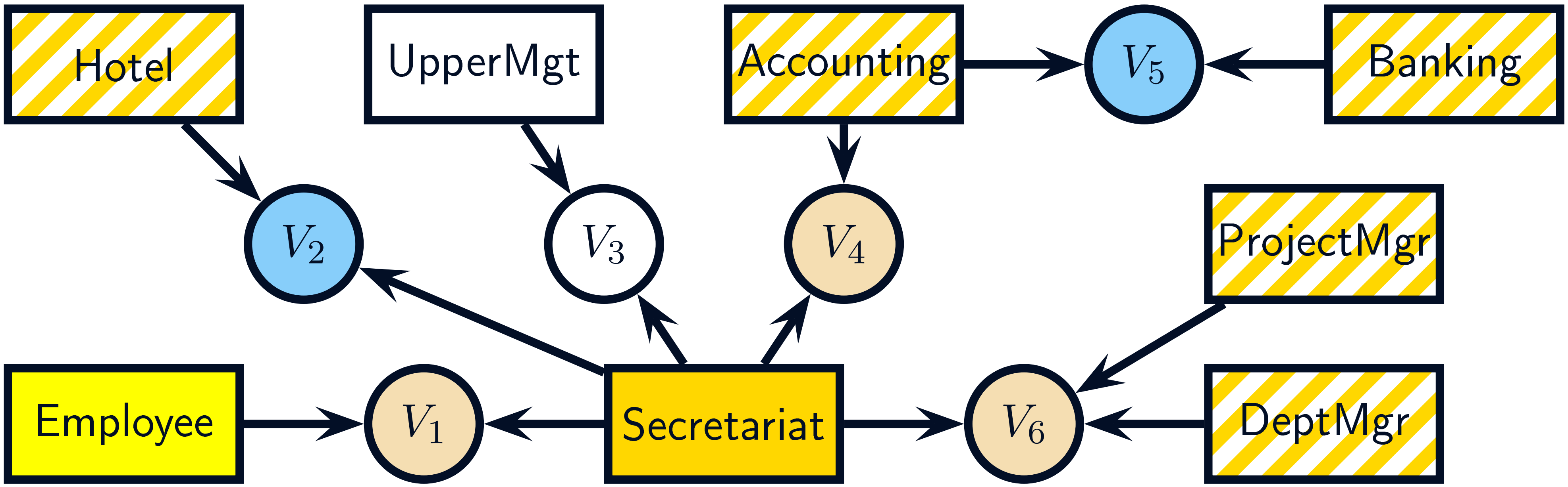


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

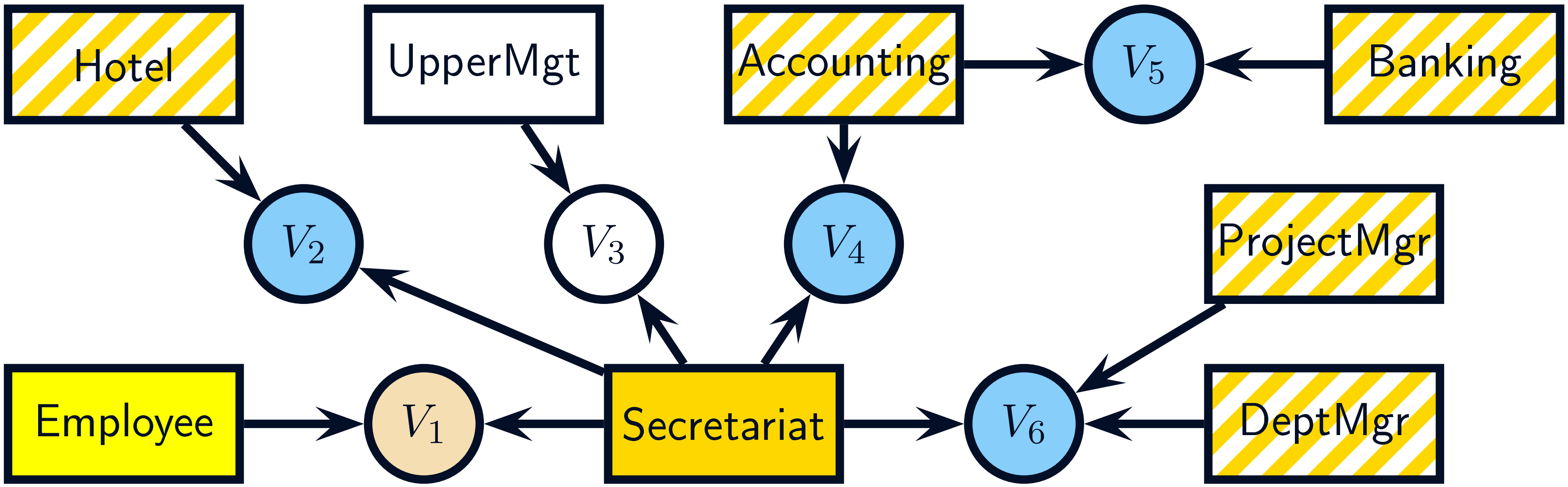


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

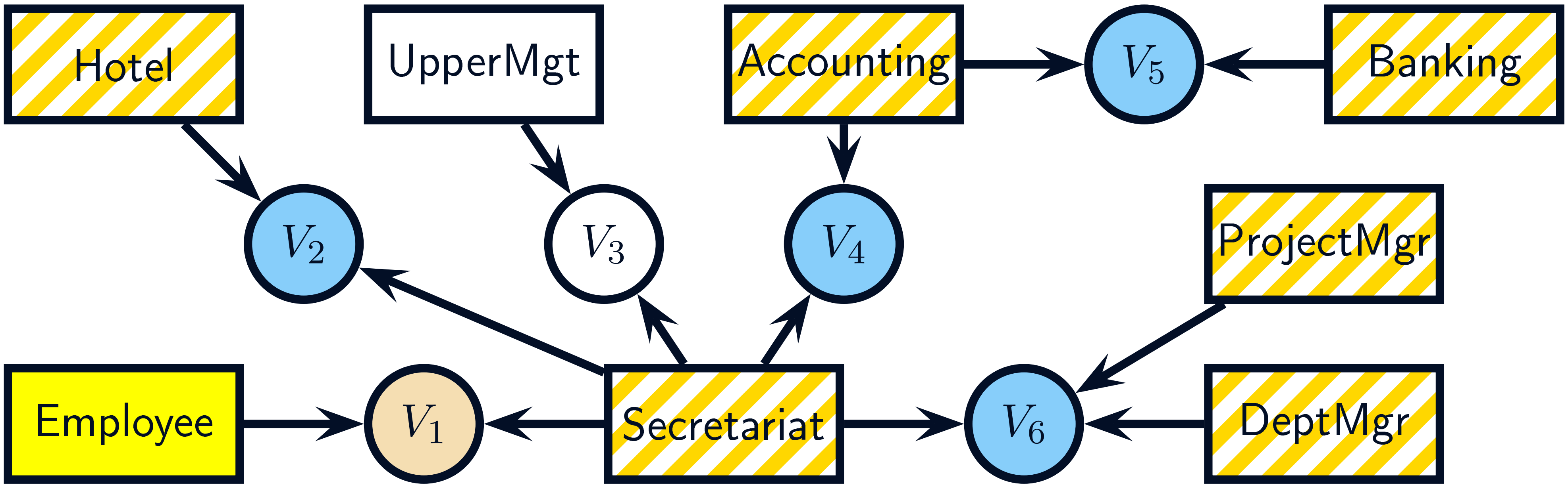


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

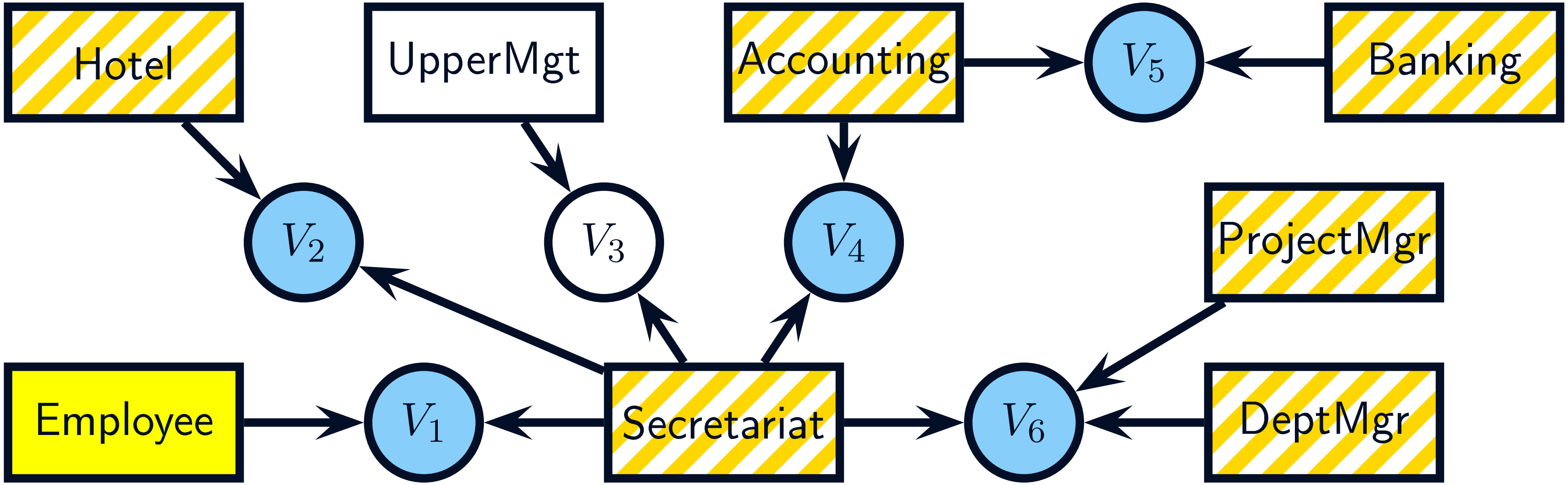


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

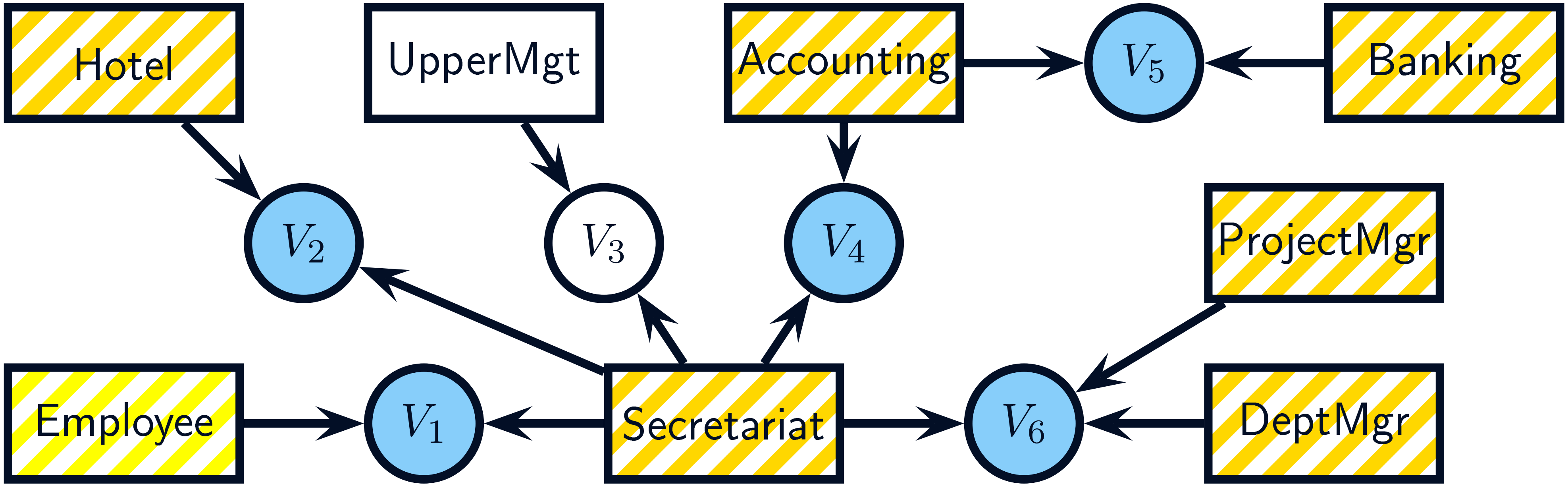


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

The Three-Stage Update-Negotiation Process

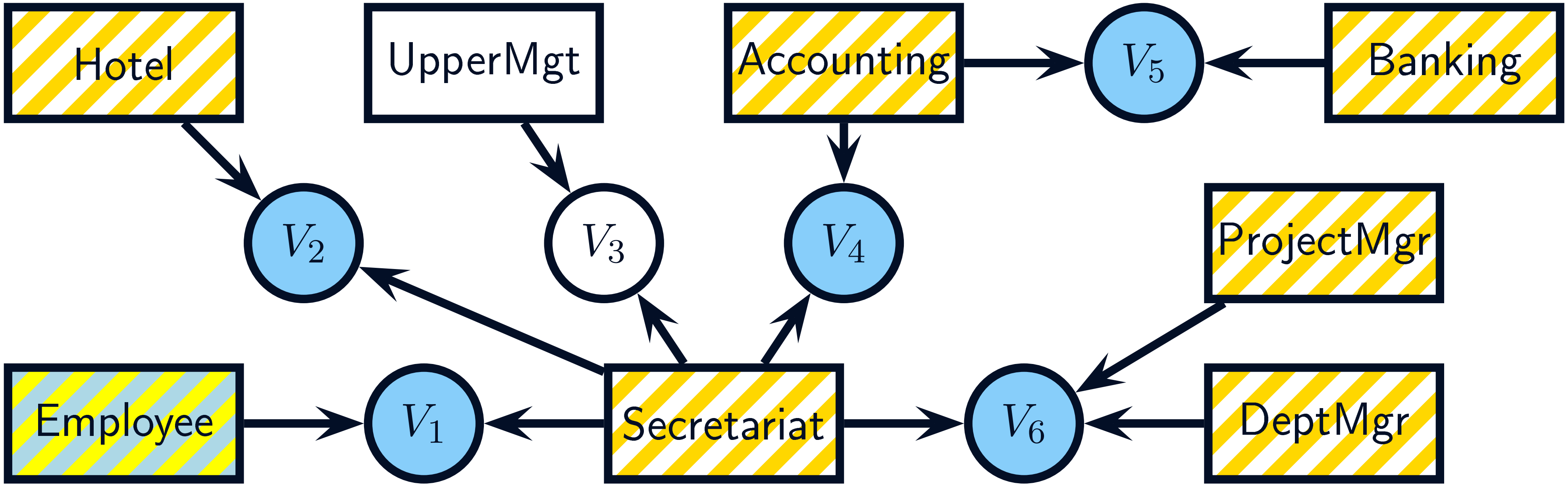


Stage 1: The initial update is passed, step-by-step, to the inner ports of neighboring components, following a project-lift cycle.

Stage 2: The decisions of *extremal components* are passed back to the outer ports of neighboring components.

- This can proceed in parallel with parts of Stage 1.
- No further decisions are made during this stage.

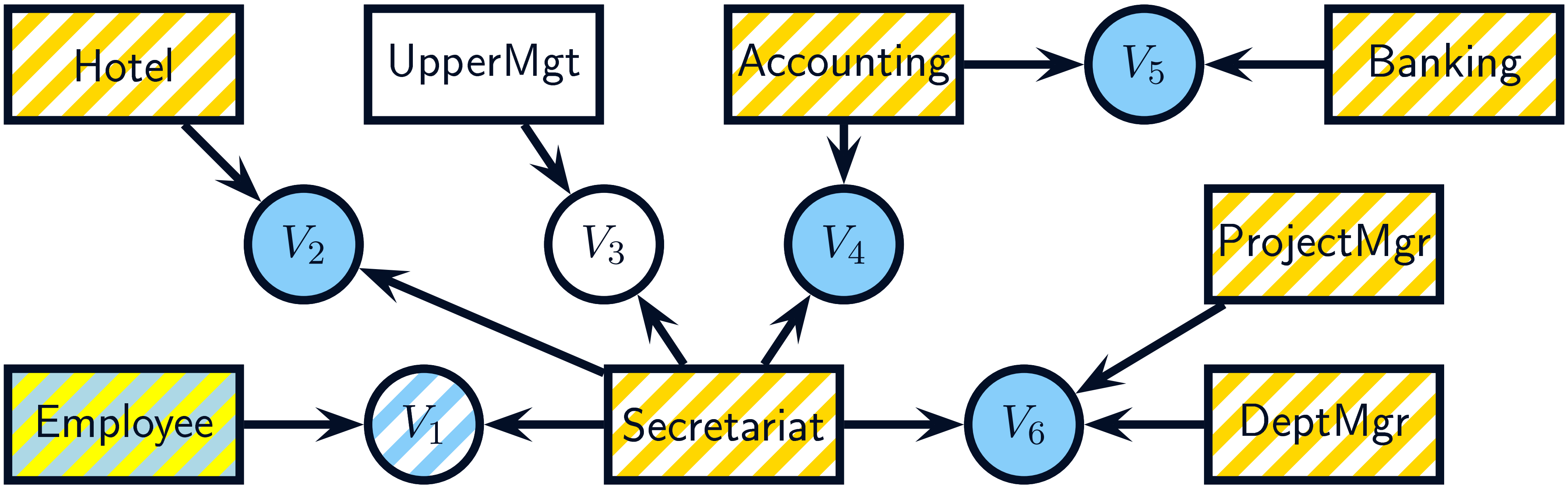
The Three-Stage Update-Negotiation Process



Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

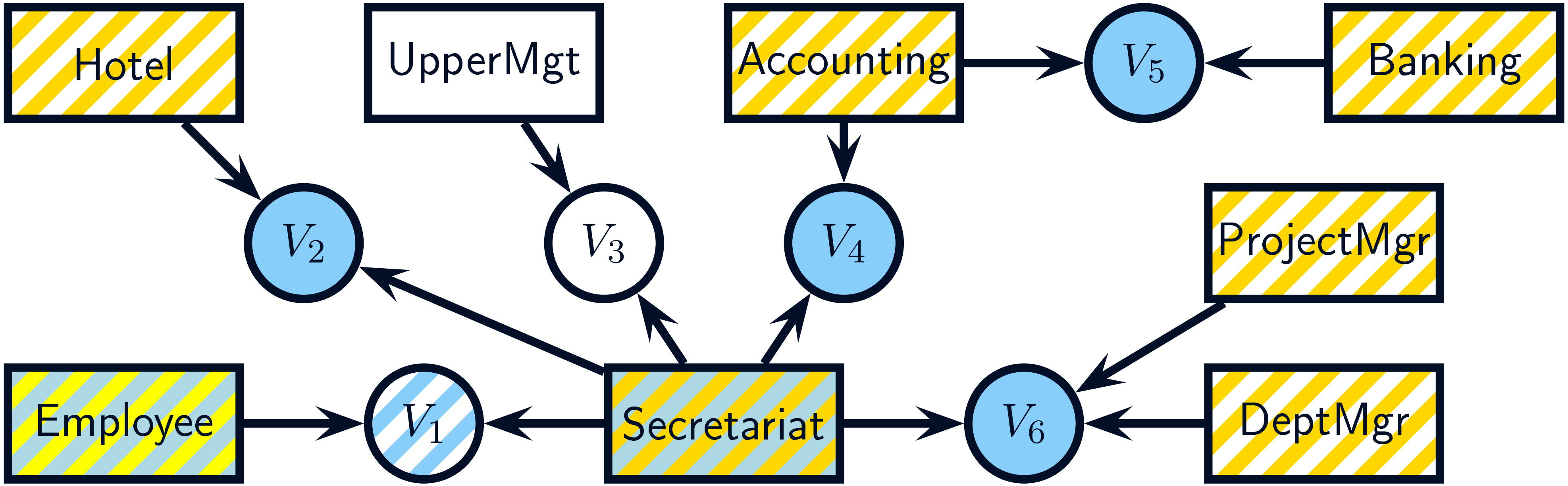
The Three-Stage Update-Negotiation Process



Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

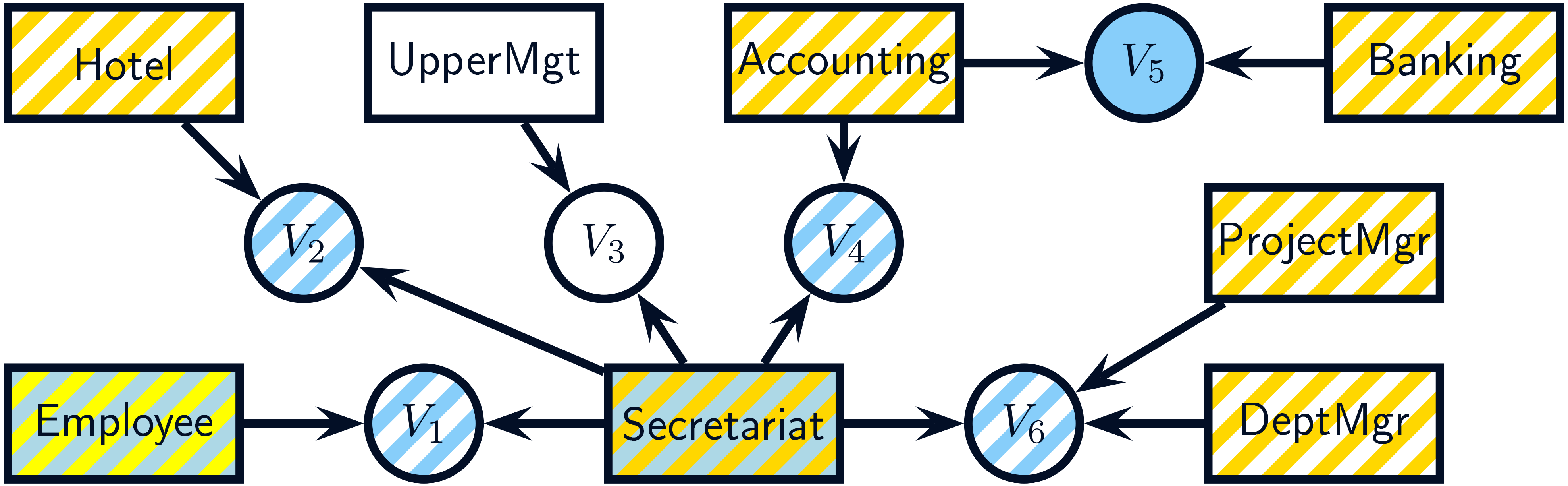
The Three-Stage Update-Negotiation Process



Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

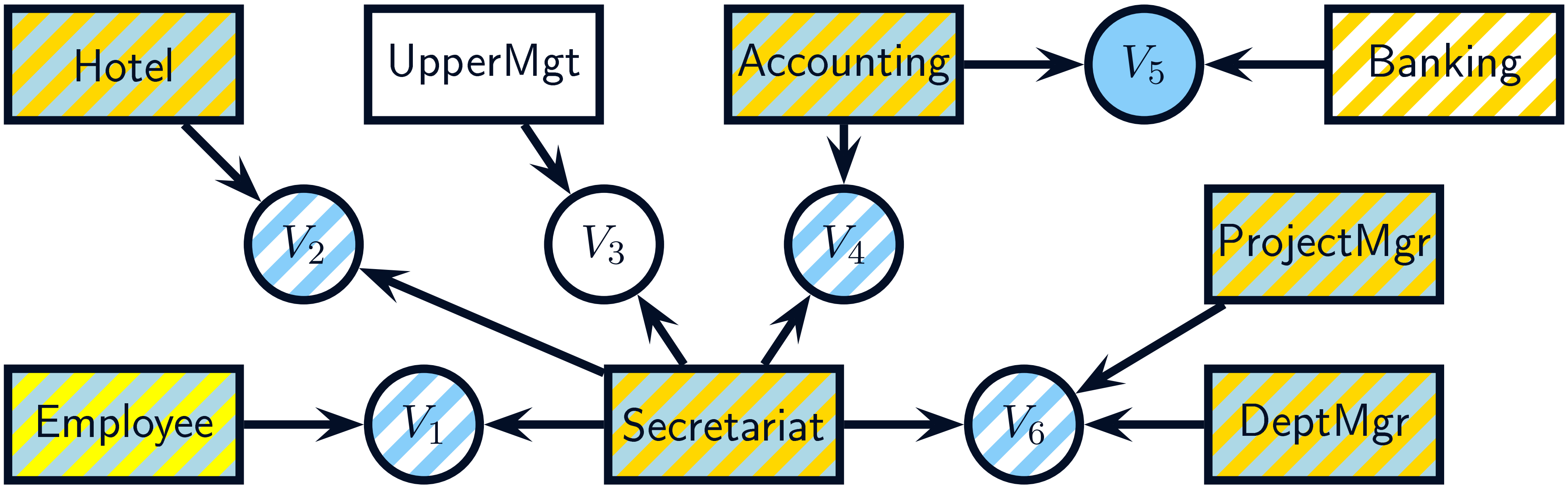
The Three-Stage Update-Negotiation Process



Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

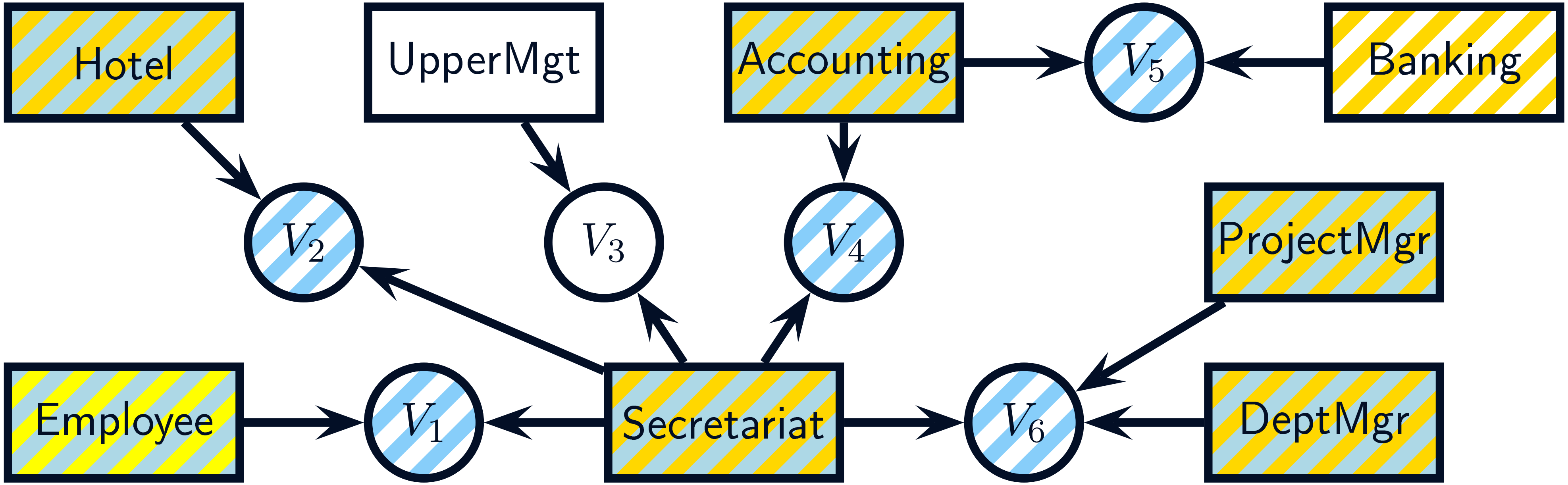
The Three-Stage Update-Negotiation Process



Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

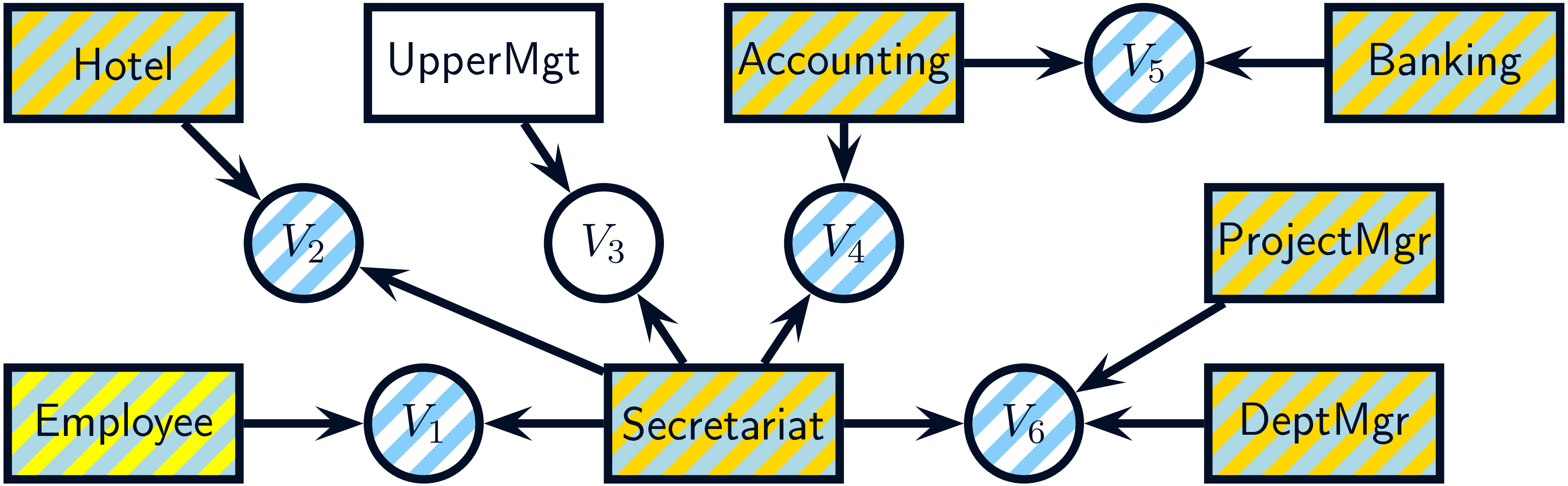
The Three-Stage Update-Negotiation Process



Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

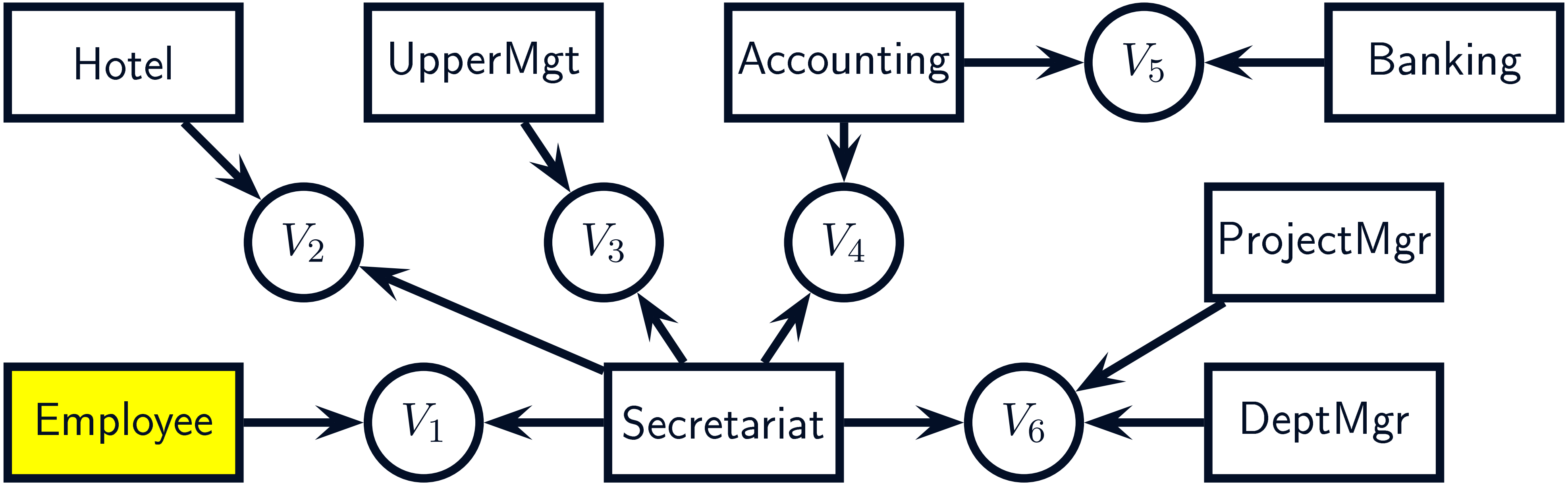
The Three-Stage Update-Negotiation Process



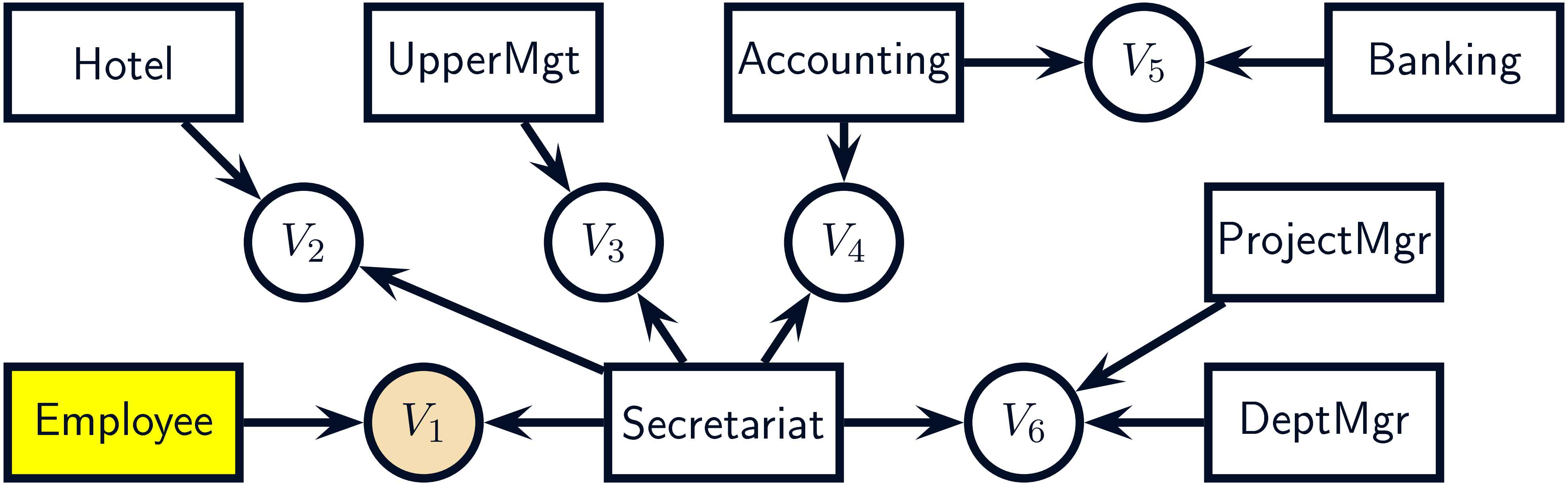
Stage 3: The initiator selects a specific update from amongst the alternatives.

- This decision is then propagated.
- There may be some choices for the other components, because even the lifting of a single update contains many possibilities.
- The result will nevertheless be a state agreed upon by all as acceptable.

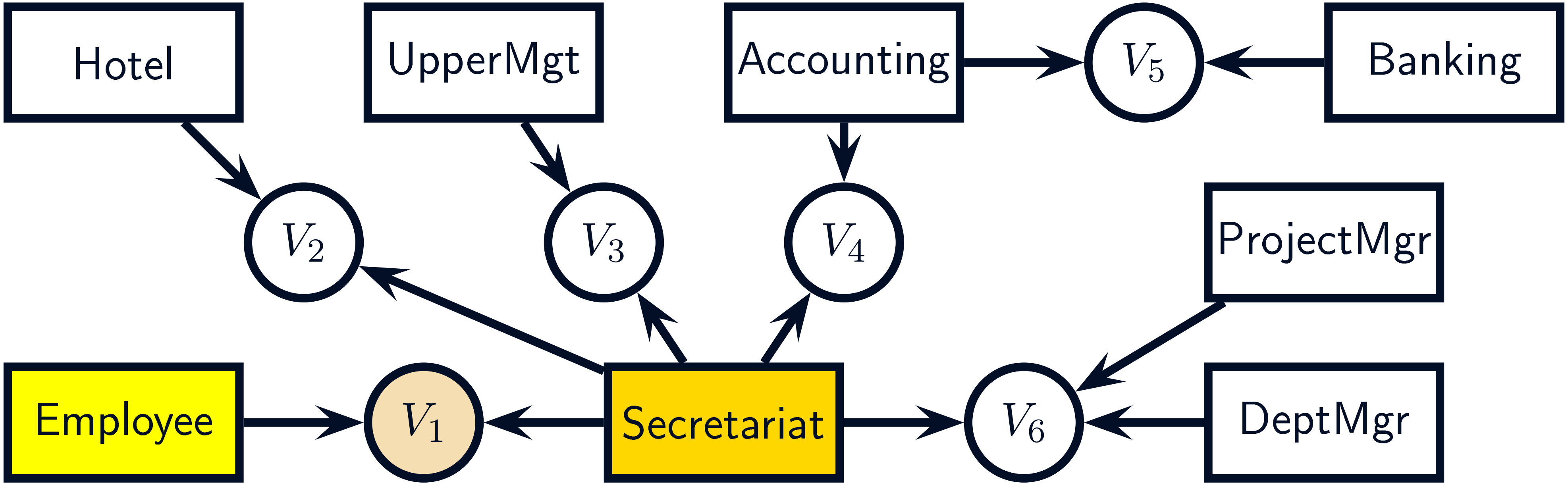
Control of Timing of Forwarding



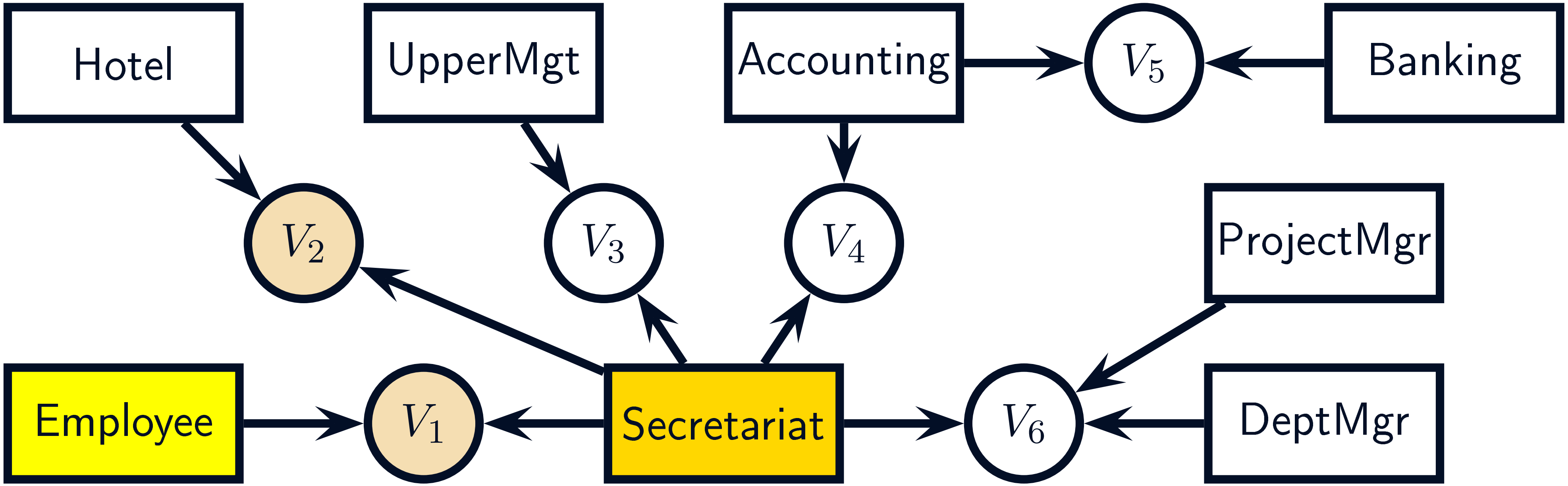
Control of Timing of Forwarding



Control of Timing of Forwarding

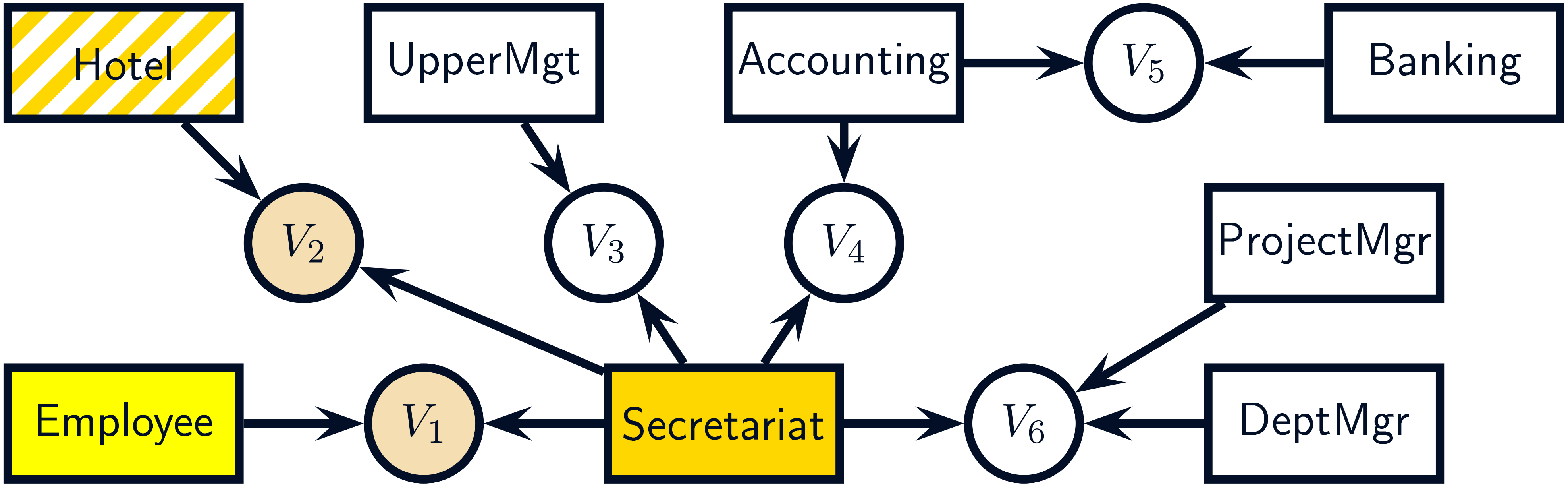


Control of Timing of Forwarding



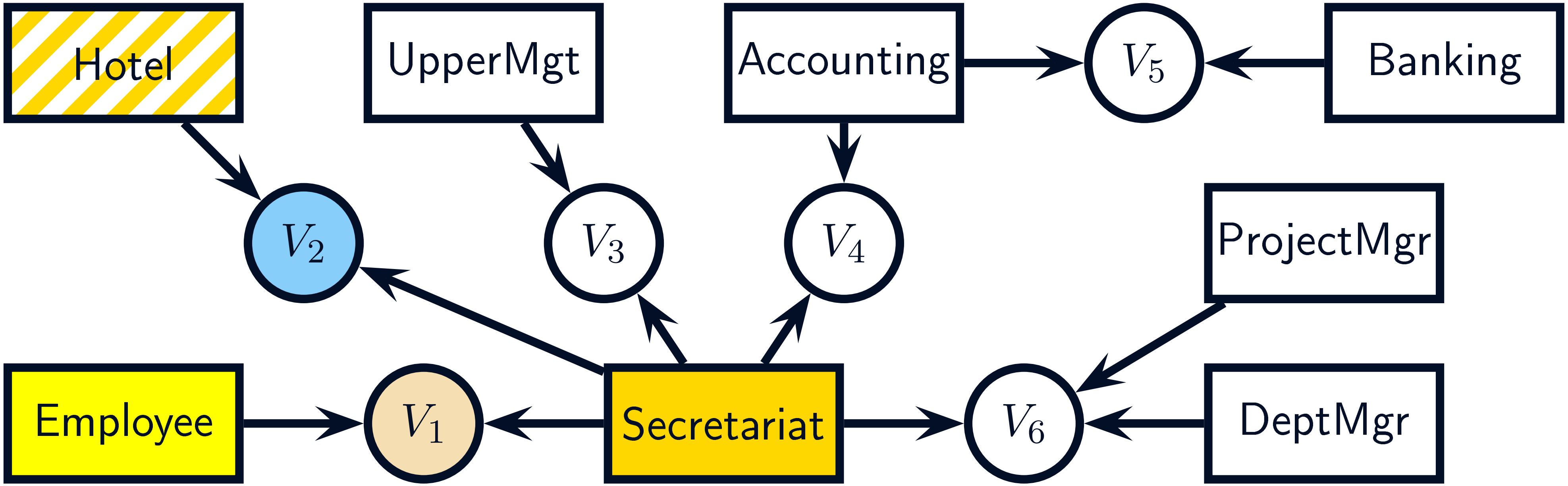
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.

Control of Timing of Forwarding



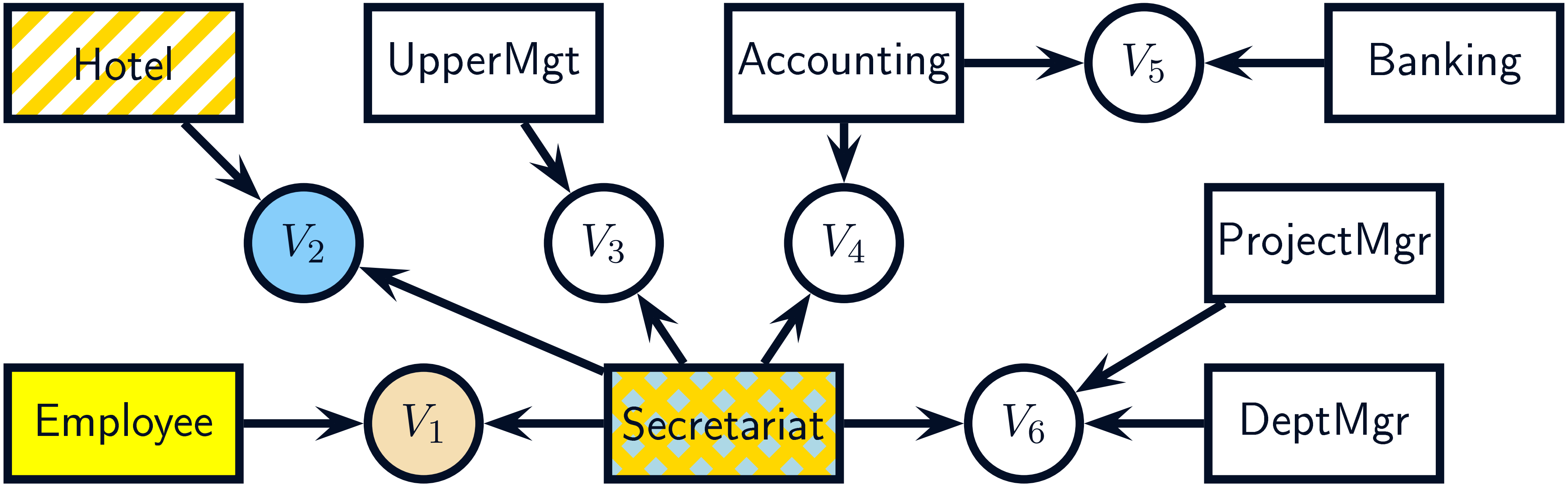
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.

Control of Timing of Forwarding



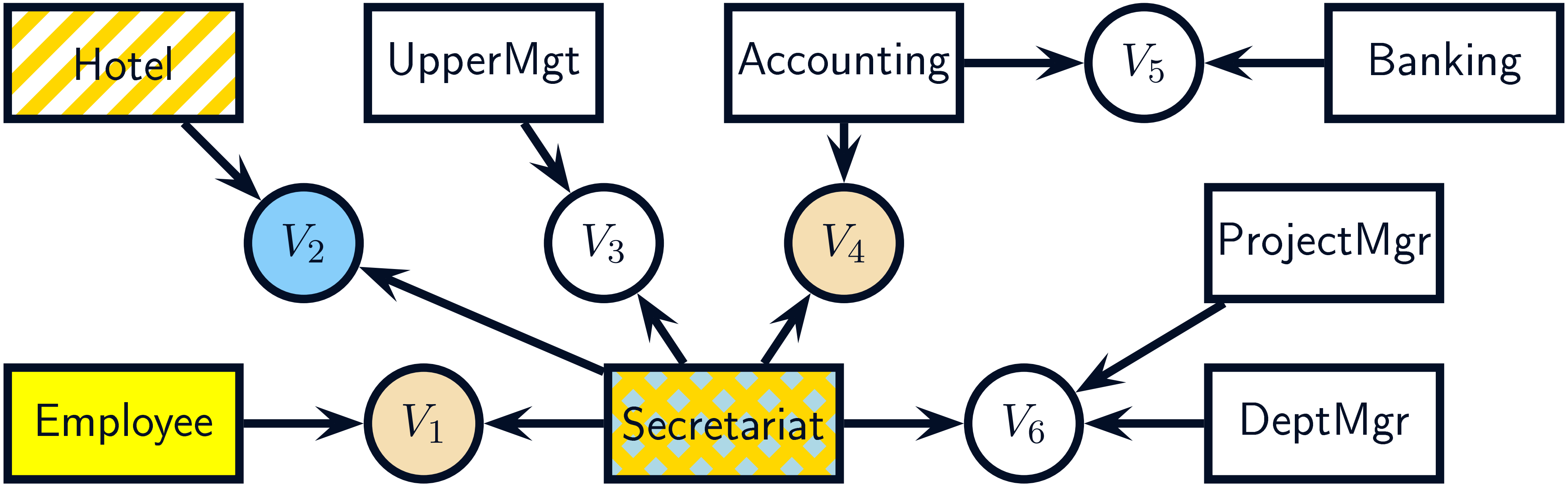
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.

Control of Timing of Forwarding



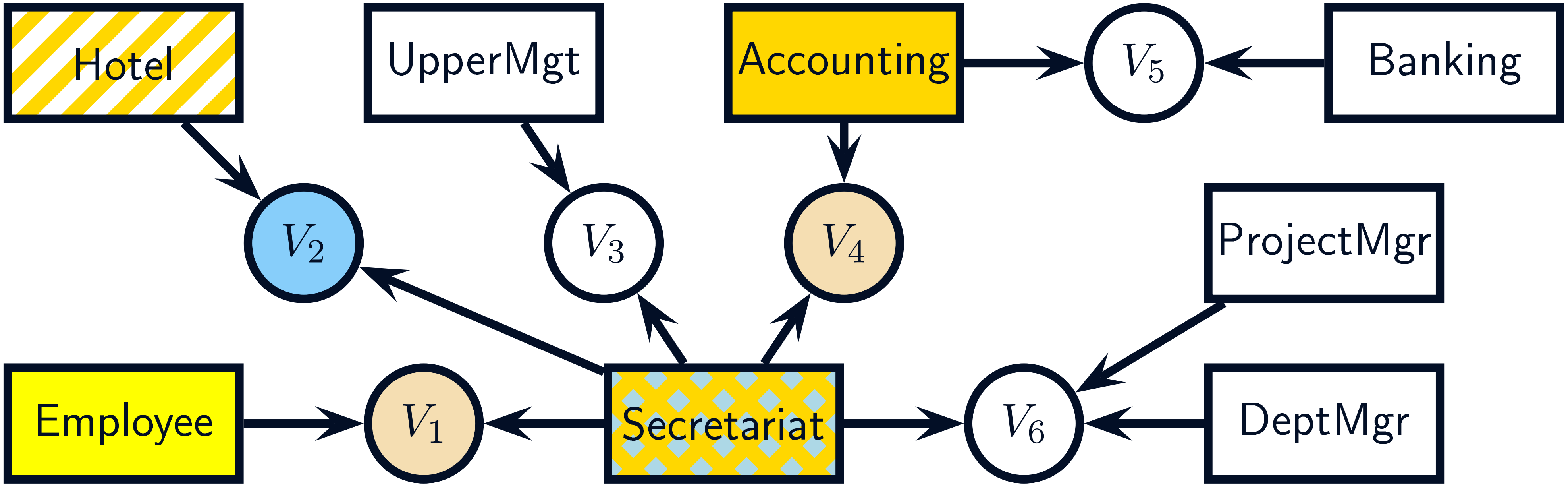
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.

Control of Timing of Forwarding



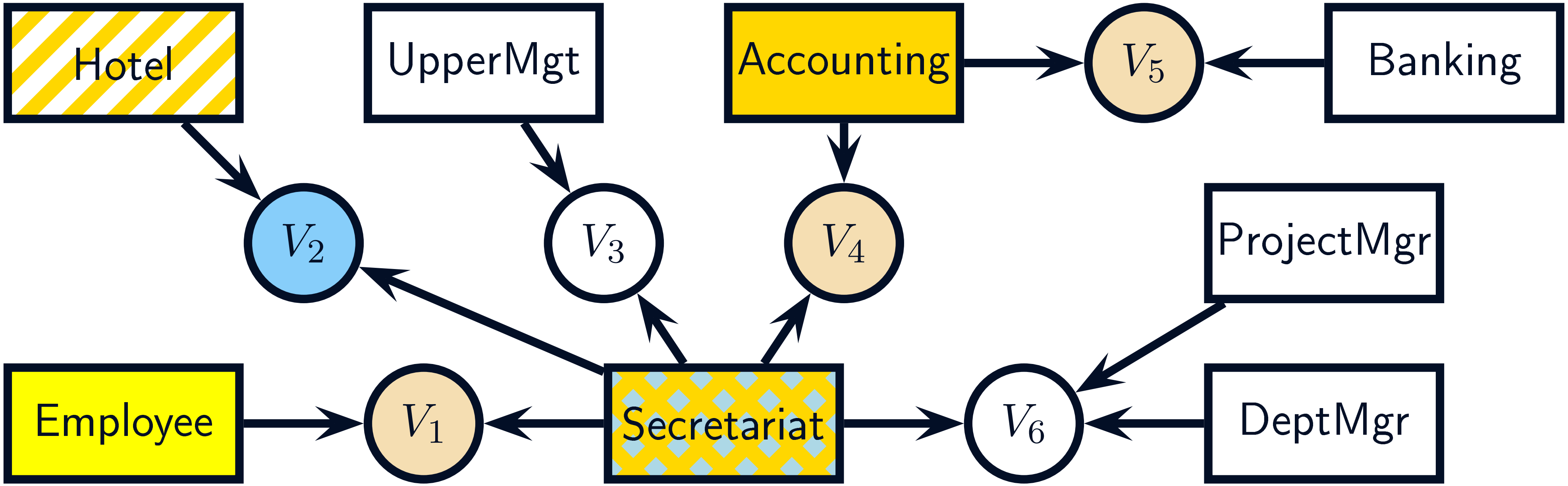
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



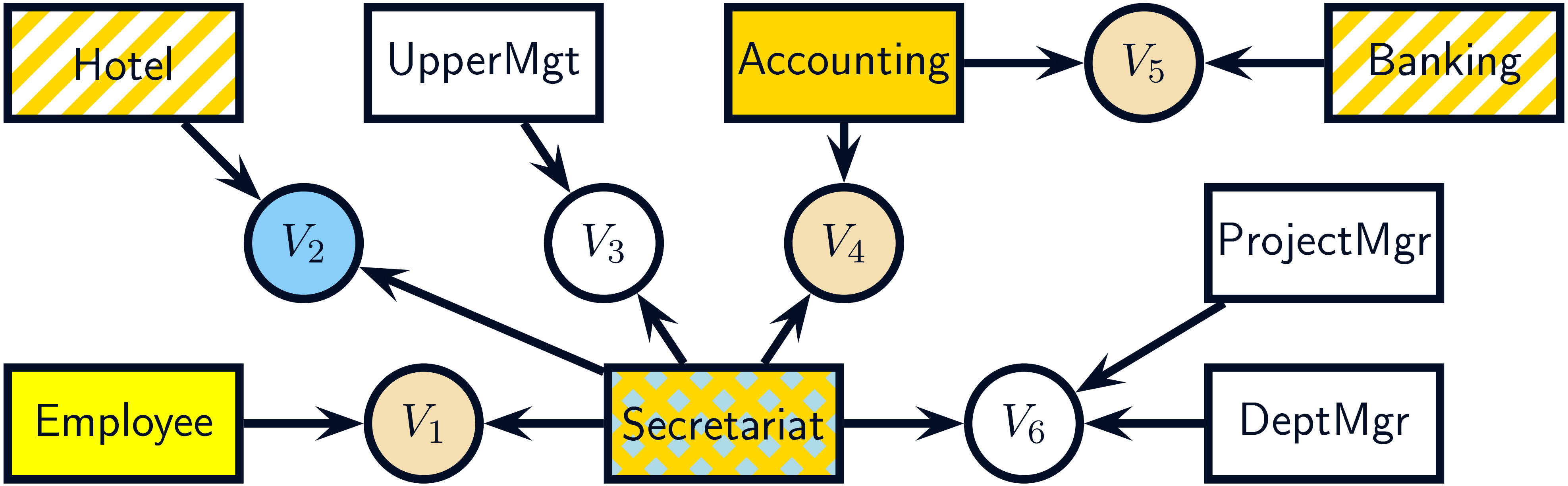
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



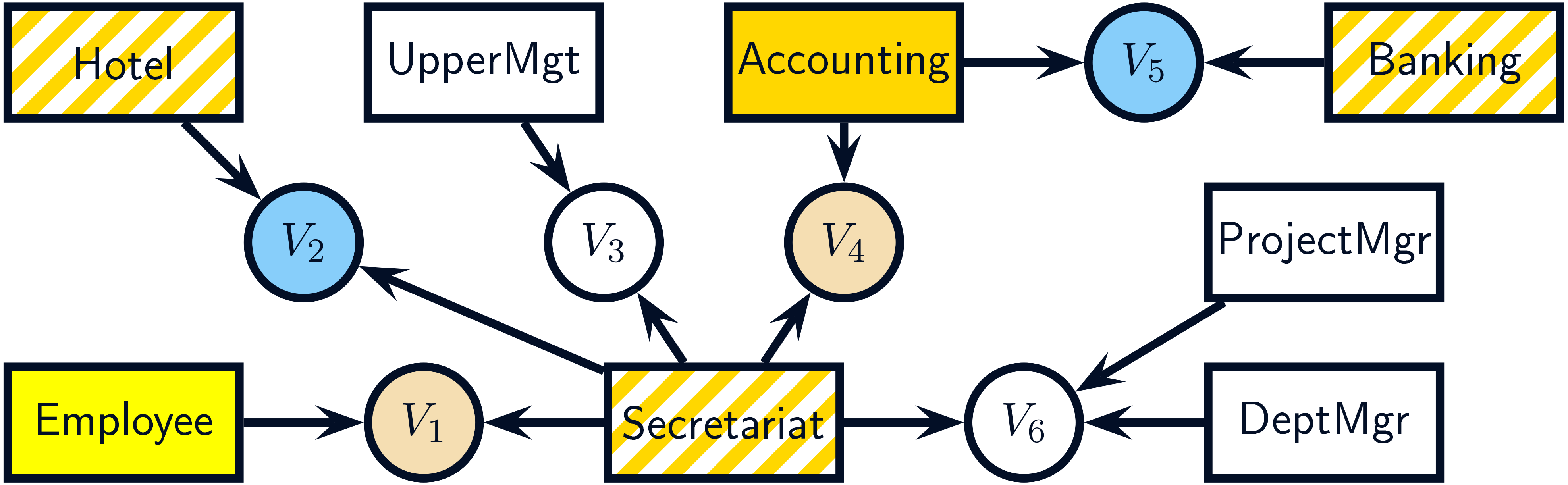
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



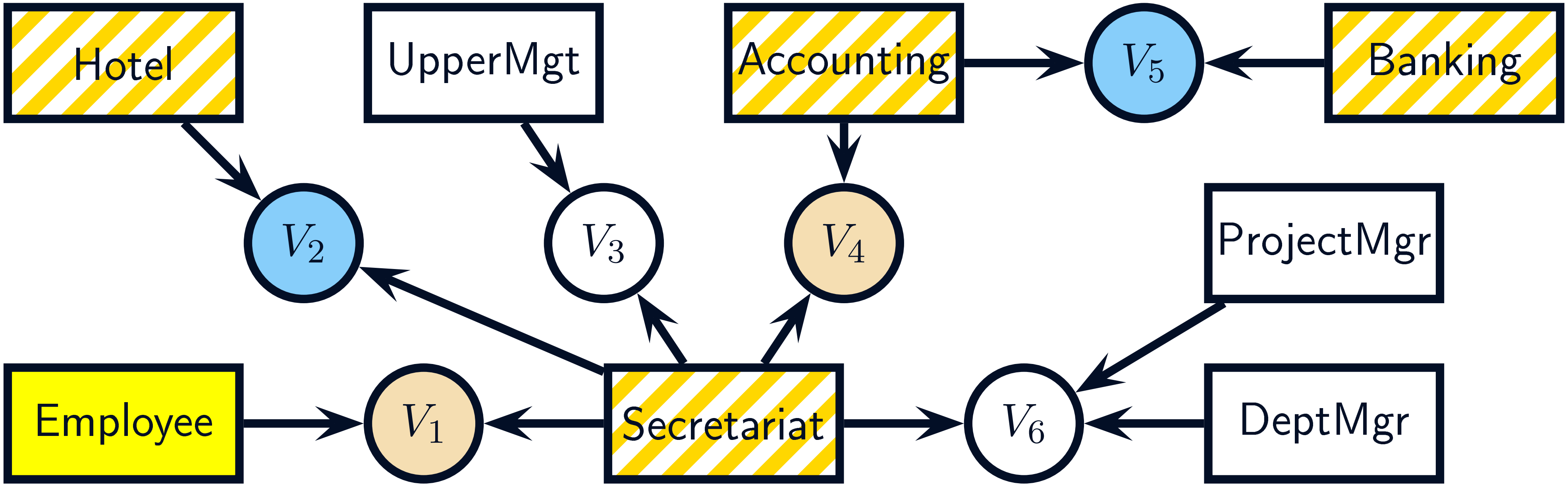
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



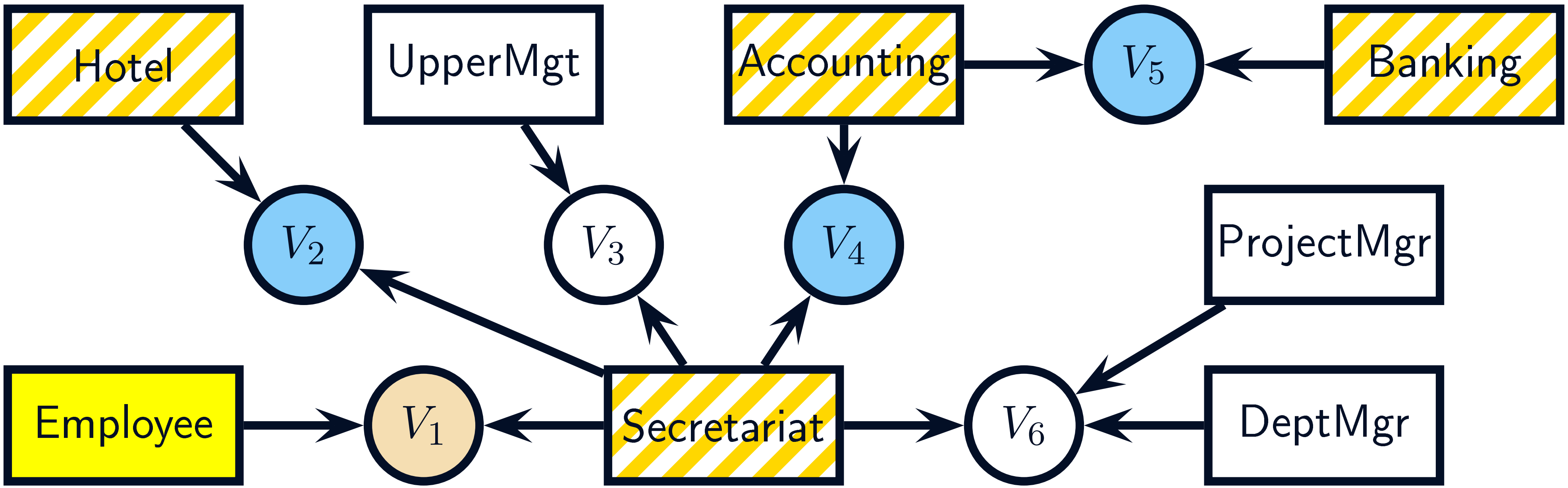
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



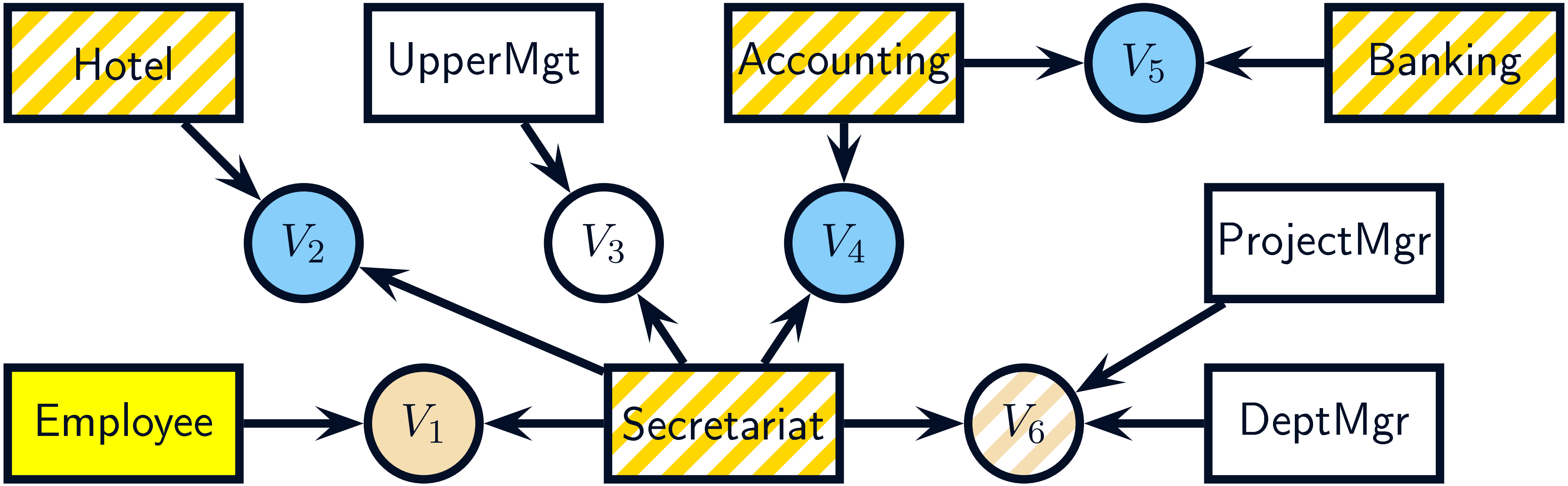
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



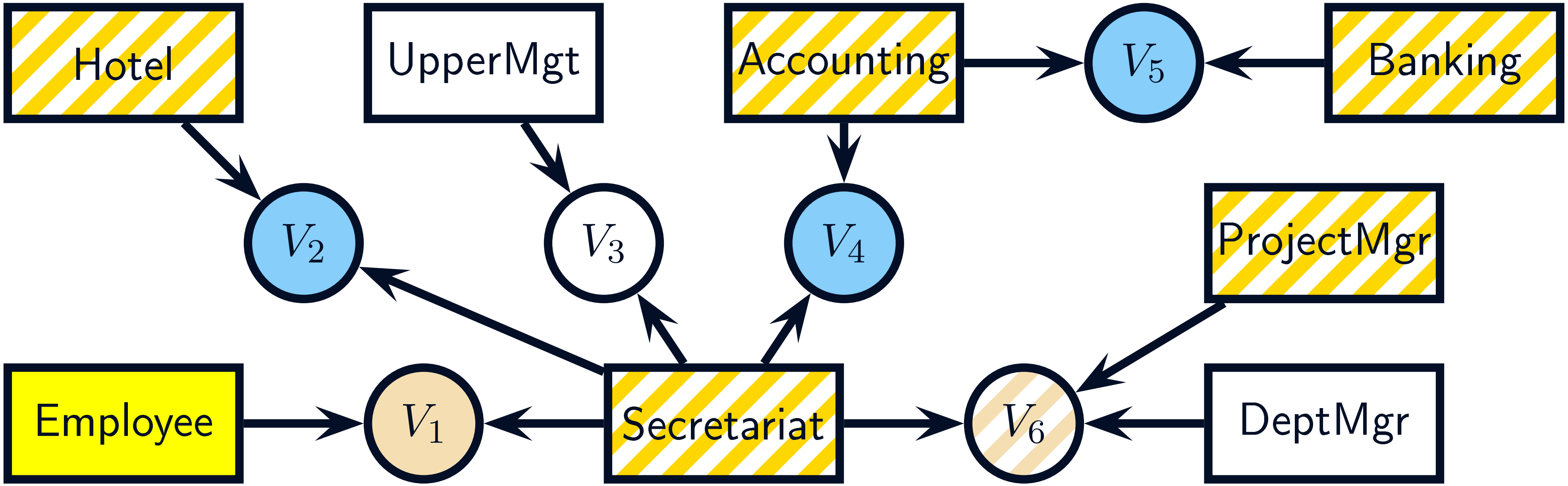
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.

Control of Timing of Forwarding



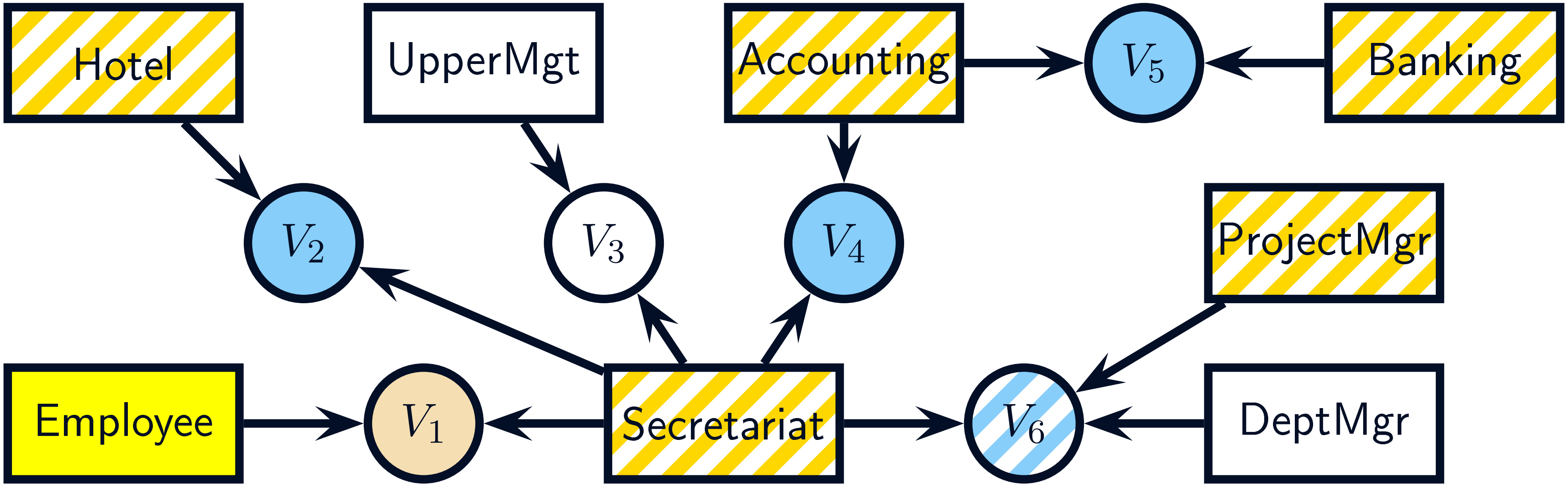
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.

Control of Timing of Forwarding

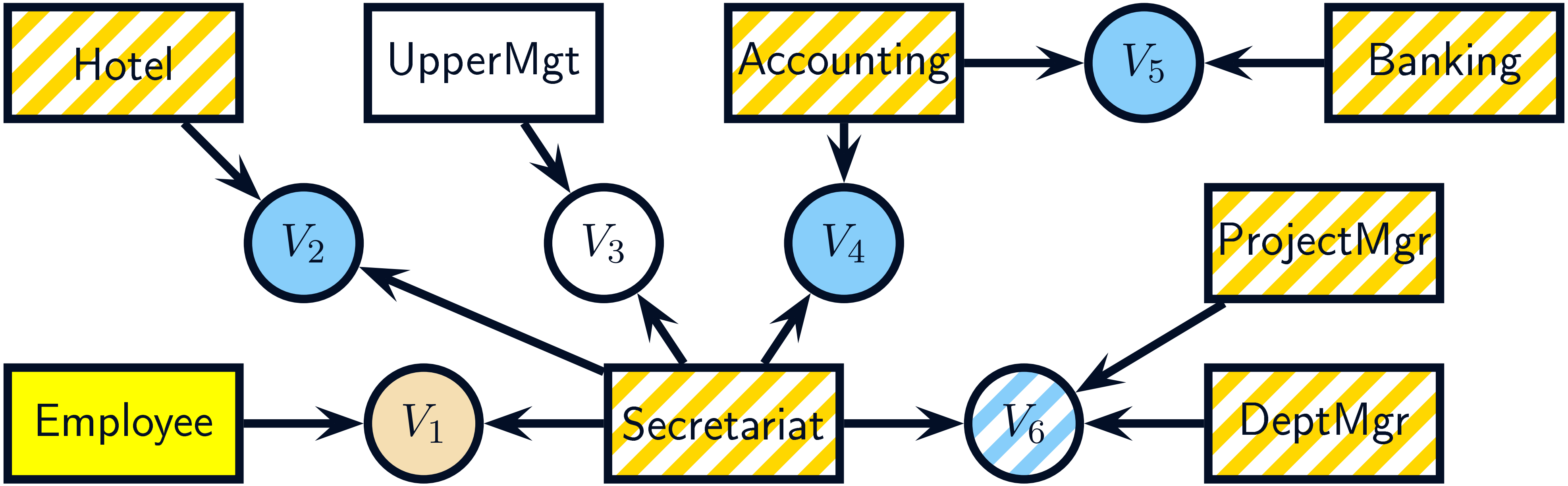


- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.

Control of Timing of Forwarding

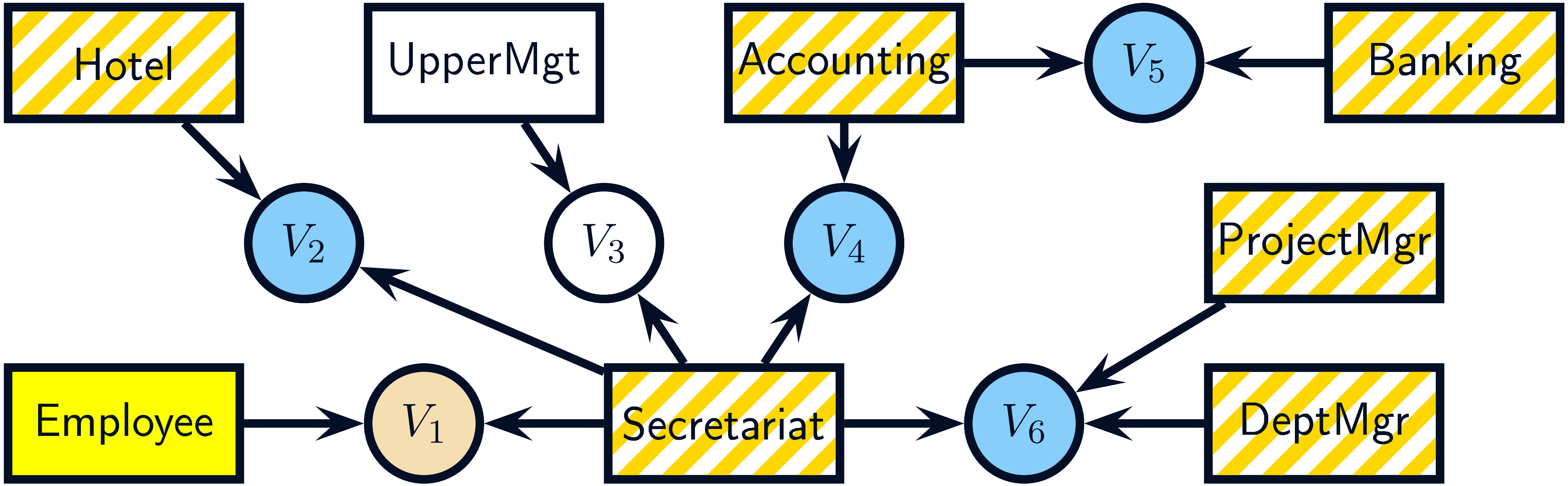


- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.



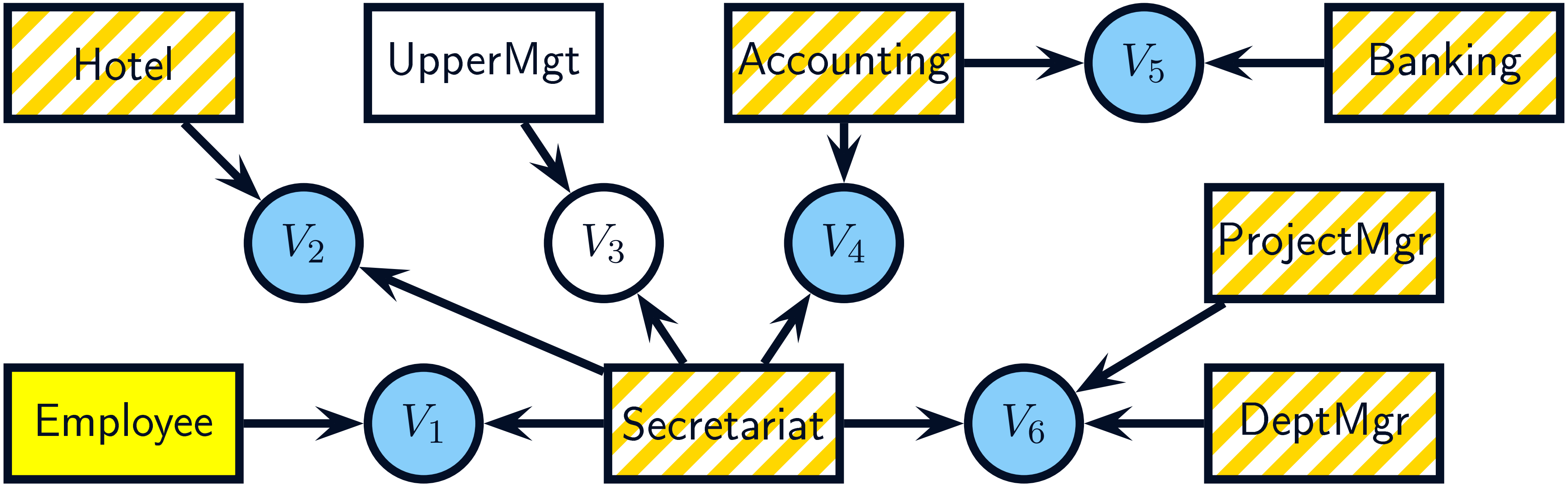
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.

Control of Timing of Forwarding



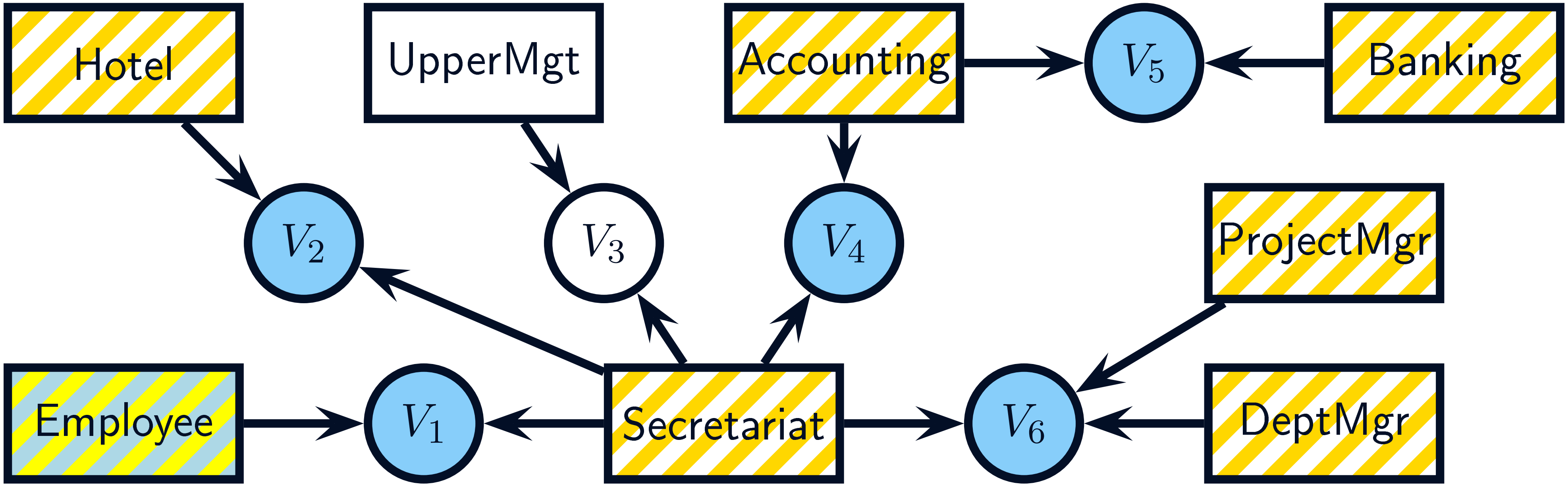
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.

Control of Timing of Forwarding



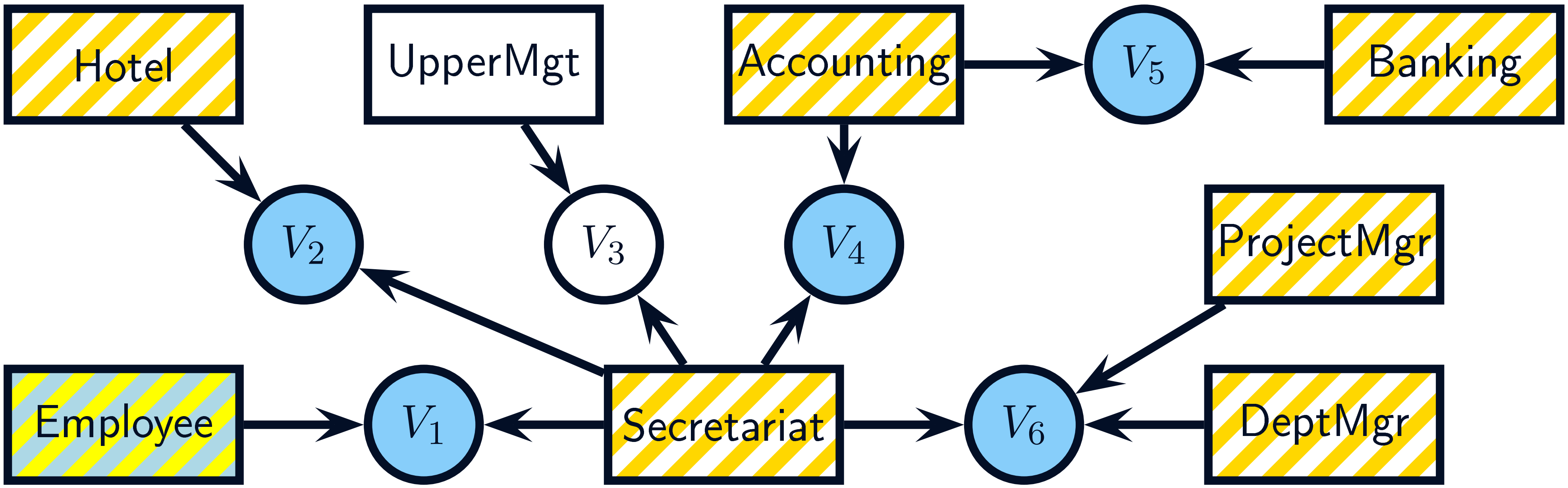
- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.

Control of Timing of Forwarding



- The *Secretariat* component may decide to forward the lifted update only to the *Hotel* at first.
- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.
- Selection of the final update proceeds as usual.

Control of Timing of Forwarding



- The proposed update which is returned allows *Secretariat* to limit its lifting before proceeding further.
- The resulting lifting may next be forwarded to *Accounting* only.
- Selective forwarding within an interconnection family is also possible.

Fact: If all lifting policies are context free, then the result is independent of the timing order. □

- The idea of granting access and update authority based upon named groups has been studied for some time.
  - *Named Protection Domains* [Baldwin 1990]
  - *Rôle-Based Access Control*

Authority Classes for Mandating Cooperation

- The idea of granting access and update authority based upon named groups has been studied for some time.
  - *Named Protection Domains* [Baldwin 1990]
  - *Rôle-Based Access Control*
- Similar ideas are used to assign update rights to users or groups of users of database components.

Authority Classes for Mandating Cooperation

- The idea of granting access and update authority based upon named groups has been studied for some time.
  - *Named Protection Domains* [Baldwin 1990]
  - *Rôle-Based Access Control*
- Similar ideas are used to assign update rights to users or groups of users of database components.
- A central part of this formalism is the *authority hierarchy*, which is just a finite, partially-ordered set (poset) which models rôles.

Authority Classes for Mandating Cooperation

- The idea of granting access and update authority based upon named groups has been studied for some time.
  - *Named Protection Domains* [Baldwin 1990]
  - *Rôle-Based Access Control*
- Similar ideas are used to assign update rights to users or groups of users of database components.
- A central part of this formalism is the *authority hierarchy*, which is just a finite, partially-ordered set (poset) which models rôles.
- It might be very simple: *Example*:

$$\text{Secretaries} < \text{Scientists} < \text{Managers}$$
$$\text{Scientists} < \text{Accountants}$$

Authority Classes for Mandating Cooperation

- The idea of granting access and update authority based upon named groups has been studied for some time.
    - *Named Protection Domains* [Baldwin 1990]
    - *Rôle-Based Access Control*
- Similar ideas are used to assign update rights to users or groups of users of database components.
- A central part of this formalism is the *authority hierarchy*, which is just a finite, partially-ordered set (poset) which models rôles.
- It might be very simple: *Example*:

$$\begin{tikzcd} & & \text{Managers} \\ \text{Secretaries} \arrow[r, dash, "<"] & \text{Scientists} \arrow[ur, dash, "<"] \arrow[dr, dash, "<"'] & \\ & & \text{Accountants} \end{tikzcd}$$

- However, more complex rôles are easily envisaged; for example, *Travel_Managing_Secretary* who would be given rights to manage travel requests, but nothing else.

Access Rights for Database Components

- There are four classes of access rights.

Access Rights for Database Components

- There are four classes of access rights.
  - The right to initiate an update.
  - The right to limit an update family during lifting.
  - The right to order in time the forwarding of update requests to distinct ports.
  - The right to order in time the forwarding of update requests to distinct components connected to the same port.

The Right to Initiate an Update Proposal

Form: Initiate(Component : C, View : Γ, Authority_class : A).

The Right to Initiate an Update Proposal

Form: $\text{Initiate}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

- A specifies the authority class which is assigned to the update proposal.

The Right to Initiate an Update Proposal

Form: Initiate(Component : C,   View : Γ,   Authority_class : A).

- A specifies the authority class which is assigned to the update proposal.
- Γ is a view which restricts the set of possible updates.

The Right to Initiate an Update Proposal

Form: $\text{Initiate}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A).$

- A specifies the authority class which is assigned to the update proposal.
- Γ is a view which restricts the set of possible updates.
- The restriction is that the update must be entirely visible within that view.

The Right to Initiate an Update Proposal

Form: $\text{Initiate}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

- A specifies the authority class which is assigned to the update proposal.
- Γ is a view which restricts the set of possible updates.
- The restriction is that the update must be entirely visible within that view.
- In the relational context, Γ would typically be a restriction.

The Right to Initiate an Update Proposal

Form: Initiate(Component : C,   View : Γ,   Authority_class : A).

- A specifies the authority class which is assigned to the update proposal.
- Γ is a view which restricts the set of possible updates.
- The restriction is that the update must be entirely visible within that view.
- In the relational context, Γ would typically be a restriction.
- The formalization of "entirely visible" is based upon semantically bijective mappings [Hegner SDKB 08].

The Right to Trim an Update Proposal

Form: Trim(Component : C, View : Γ, Authority_class : A).

The Right to Trim an Update Proposal

Form: $\text{Trim}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

- *Trimming* is the process in which a component is not required to use the minmax lifting for an input on its inner port.

The Right to Trim an Update Proposal

Form: $\mathsf{Trim}(\mathsf{Component} : C,\quad \mathsf{View} : \Gamma,\quad \mathsf{Authority\_class} : A)$.

- *Trimming* is the process in which a component is not required to use the minmax lifting for an input on its inner port.
- Rather, it may make “real” decisions about which liftings to allow.

The Right to Trim an Update Proposal

Form: $\text{Trim}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A).$

- *Trimming* is the process in which a component is not required to use the minmax lifting for an input on its inner port.
- Rather, it may make “real” decisions about which liftings to allow.
- For example, a manager may decide to allow Lena to travel to ADBIS but not to DEXA.

The Right to Trim an Update Proposal

Form: $\text{Trim}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

- *Trimming* is the process in which a component is not required to use the minmax lifting for an input on its inner port.
- Rather, it may make "real" decisions about which liftings to allow.
- For example, a manager may decide to allow Lena to travel to ADBIS but not to DEXA.
- Trimming is allowed if the authority class of the incoming update family is below (less than) A in the authority hierarchy.

The Right to Trim an Update Proposal

Form: $\text{Trim}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

- *Trimming* is the process in which a component is not required to use the minmax lifting for an input on its inner port.
- Rather, it may make "real" decisions about which liftings to allow.
- For example, a manager may decide to allow Lena to travel to ADBIS but not to DEXA.
- Trimming is allowed if the authority class of the incoming update family is below (less than) A in the authority hierarchy.
- Trimming is limited to those liftings which are entirely visible in Γ, just as in the case of update initiation.

Rights to Control the Timing of Forwarding

Form: $\text{Timing}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

Rights to Control the Timing of Forwarding

Form: $\text{Timing}(\text{Component} : C, \quad \text{View} : \Gamma, \quad \text{Authority\_class} : A)$.

- Timing control is allowed if the authority class of the incoming update family is below A in the authority hierarchy.

Rights to Control the Timing of Forwarding

Form: Timing(Component : C, View : Γ, Authority_class : A).

- Timing control is allowed if the authority class of the incoming update family is below A in the authority hierarchy.
- All ports Γ' to which the lifting is not transmitted immediately must satisfy $\Gamma' \leq \Gamma$.

Rights to Control the Timing of Forwarding

Form: Timing$(\text{Component} : C,\quad \text{View} : \Gamma,\quad \text{Authority_class} : A)$.

- Timing control is allowed if the authority class of the incoming update family is below A in the authority hierarchy.
- All ports Γ' to which the lifting is not transmitted immediately must satisfy $\Gamma' \leq \Gamma$.
- Variations which distinguish controlling the timing to separate ports vs. controlling the timing to distinct components in the same interconnection are also possible.