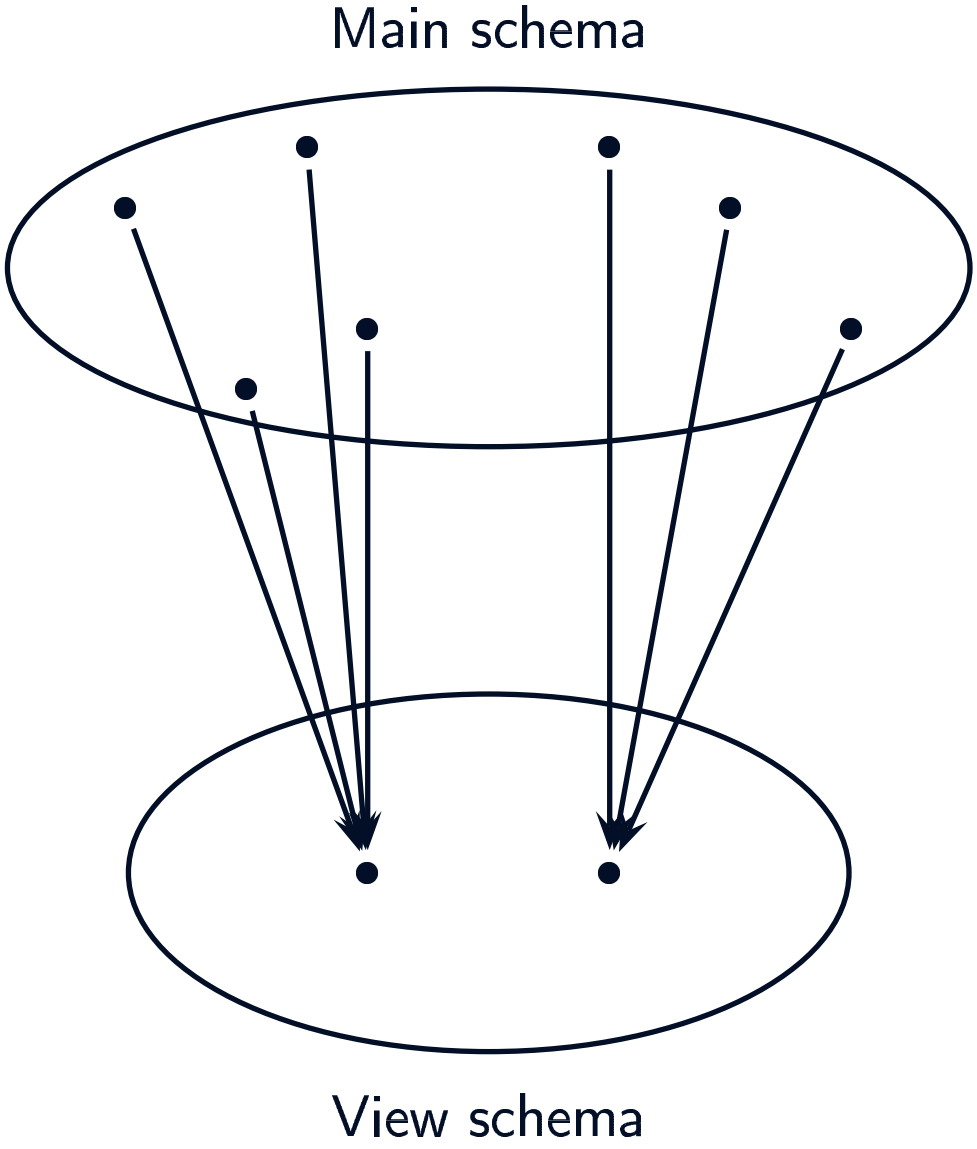
# Update Support for Database Views via Cooperation

Stephen J. Hegner
Umeå University
Department of Computing Science
Sweden

Peggy Schmidt
Christian-Albrechts-University of Kiel
Department of Computer Science
Germany

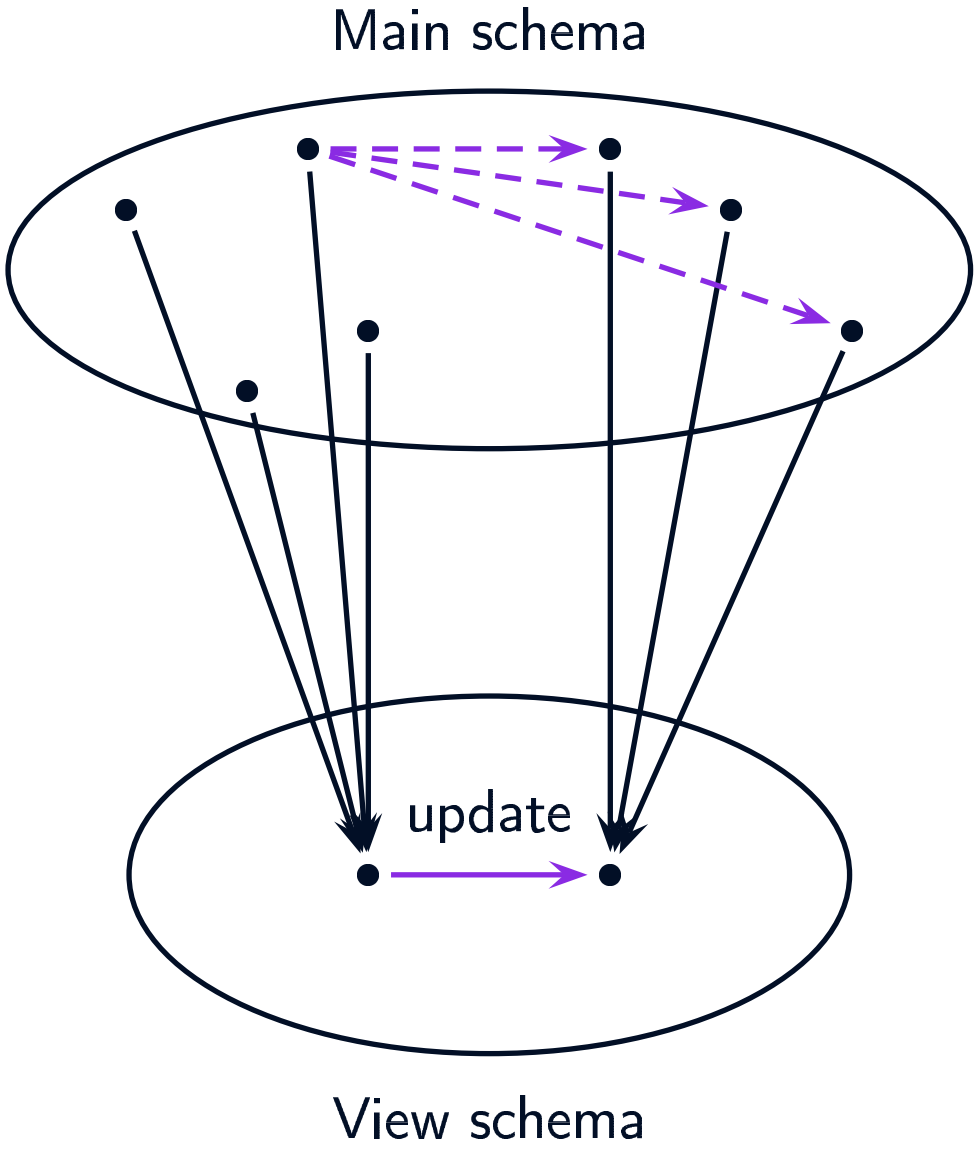
# The Update Problem for Database Views

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



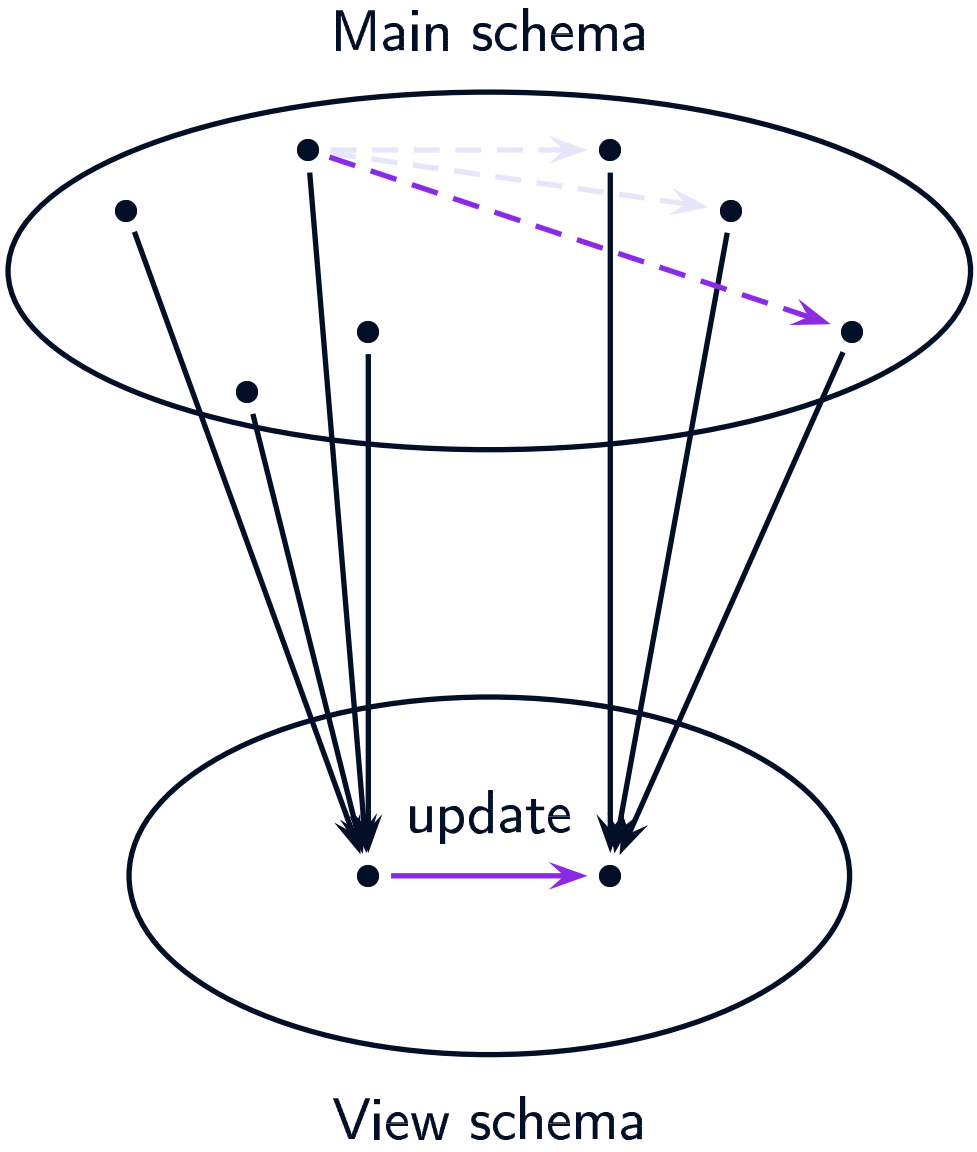
# The Update Problem for Database Views

- 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.



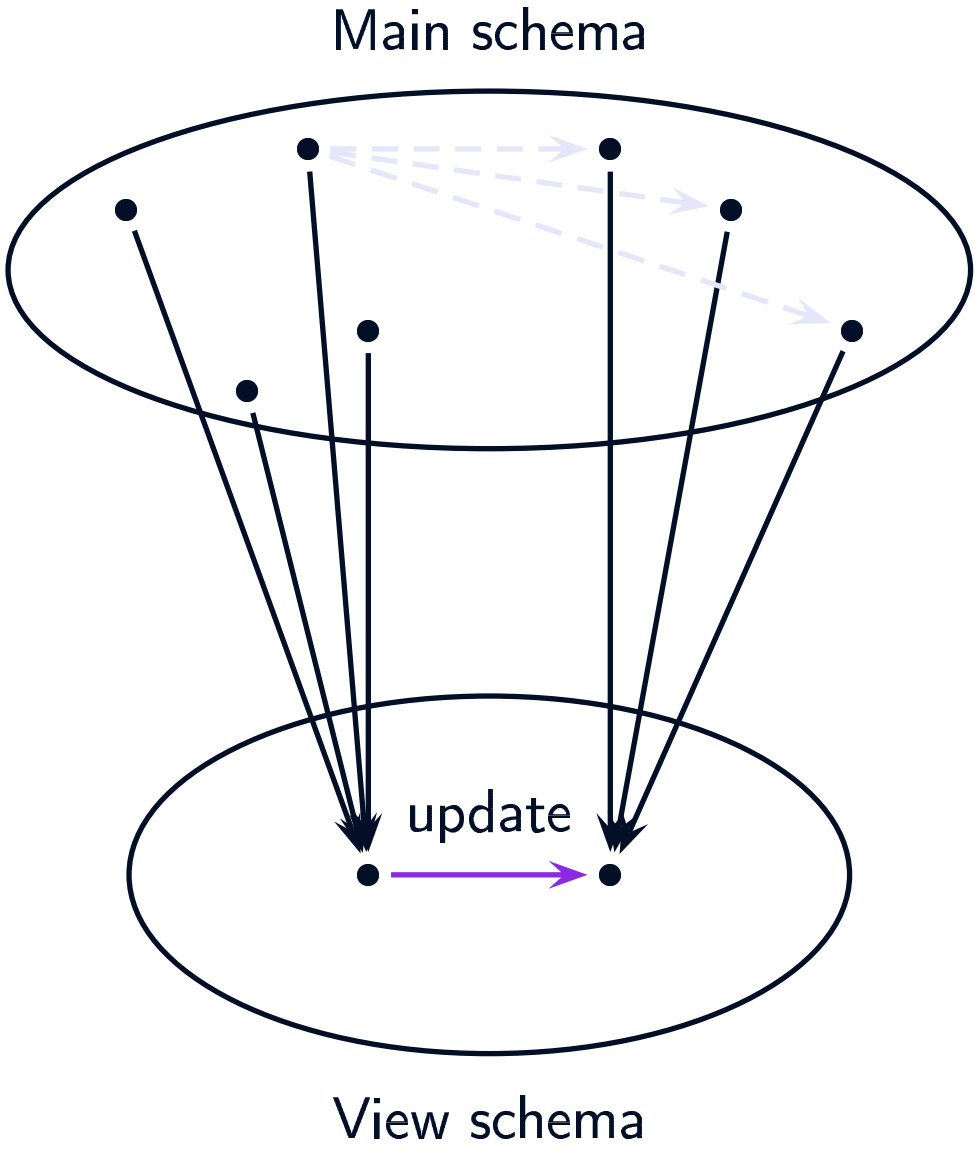
# The Update Problem for Database Views

- 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*.

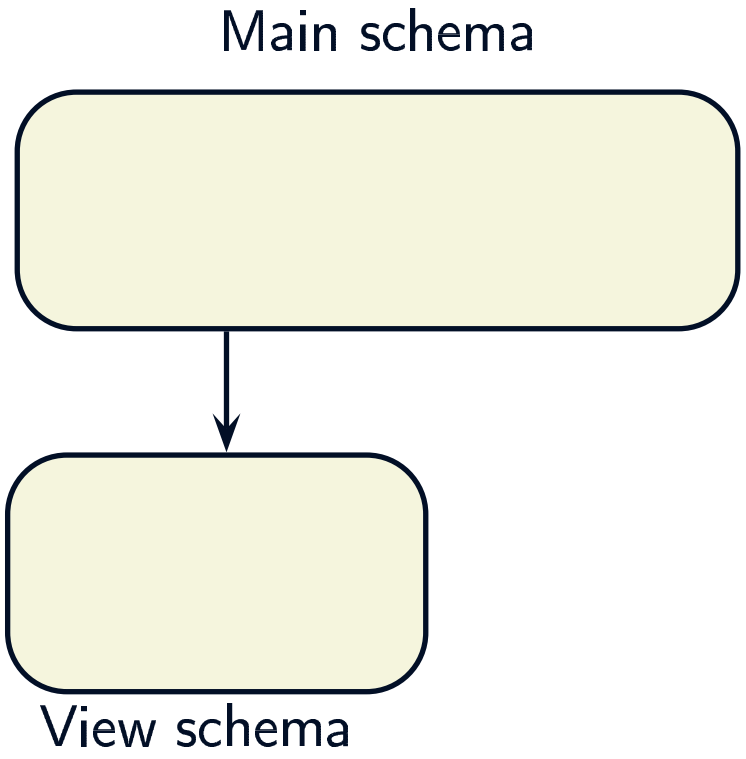


# The Update Problem for Database Views

- 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*.
- Depending upon the definition of suitability, it may not be the case that every view update has a suitable translation.

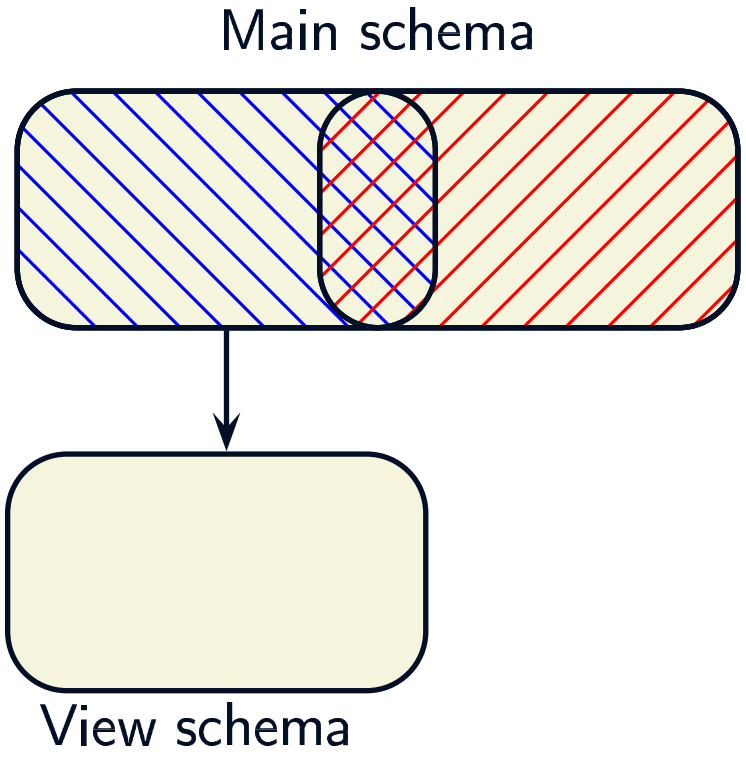


# The Constant-Complement Strategy



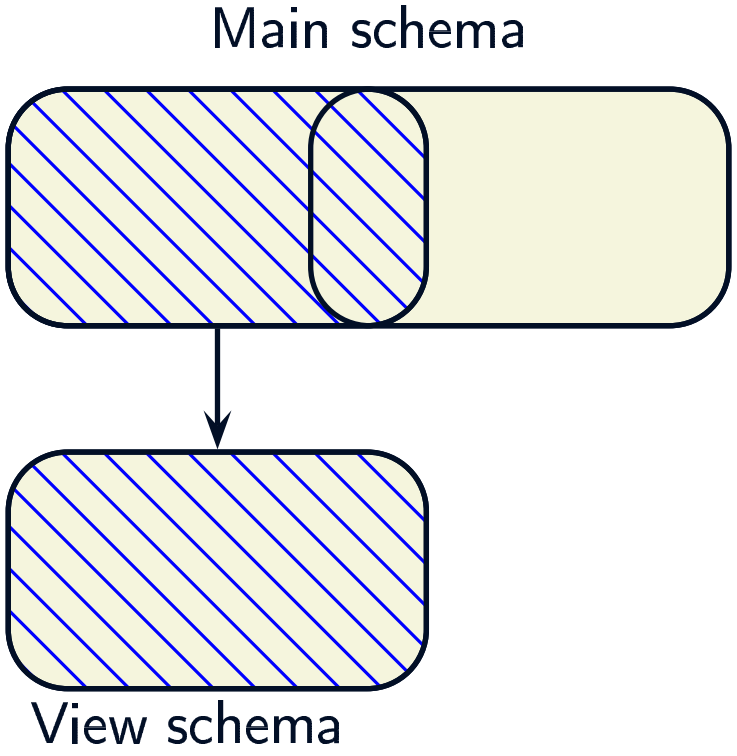
# The Constant-Complement Strategy

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



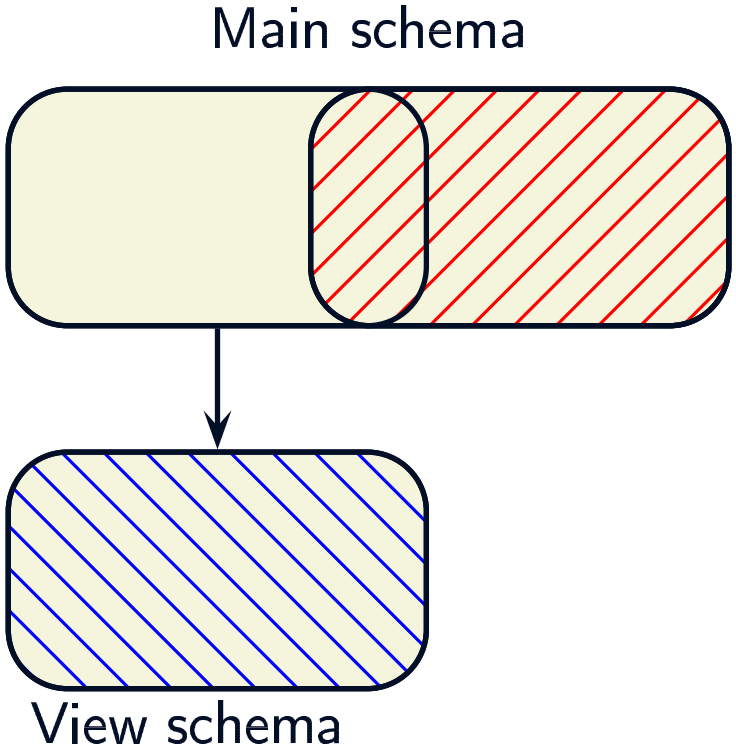
# The Constant-Complement Strategy

- 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.



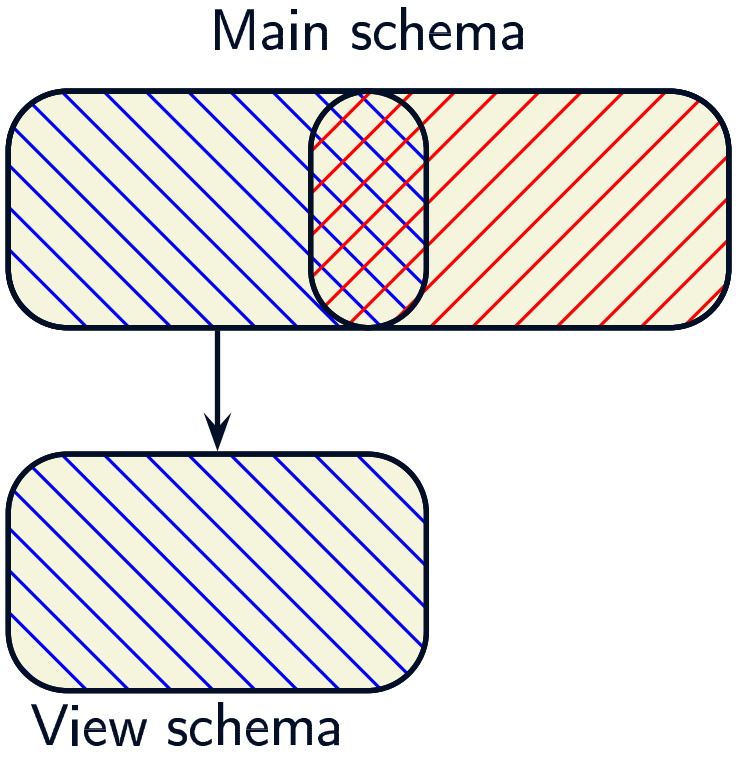
# The Constant-Complement Strategy

- 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.



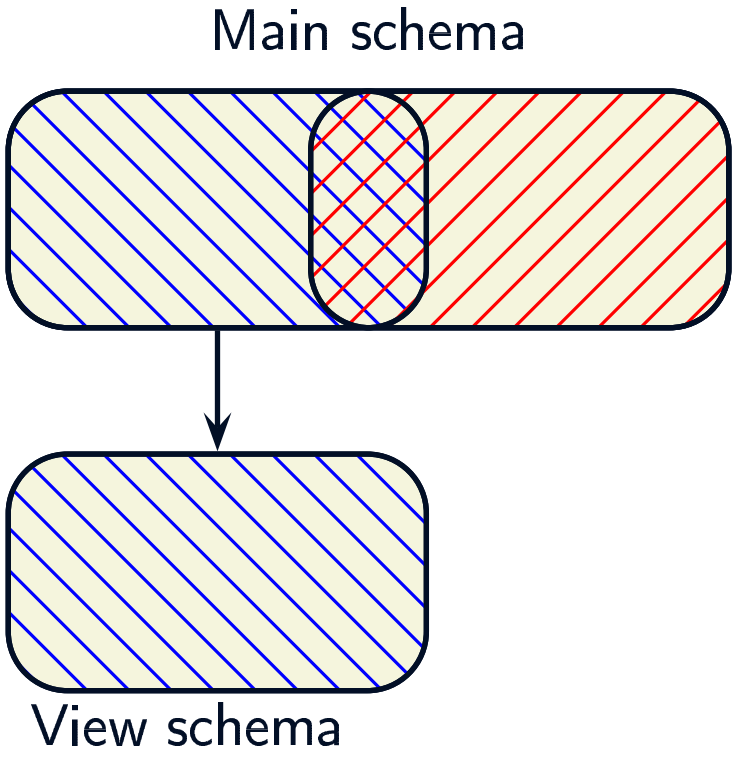
# The Constant-Complement Strategy

- 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*.



# The Constant-Complement Strategy

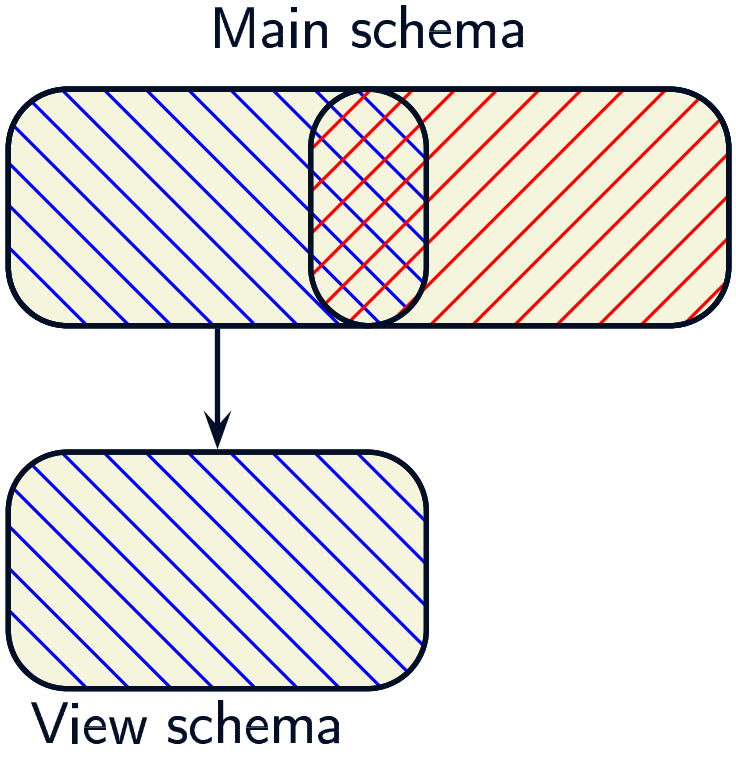
- In the constant-complement strategy [Bancilhon and Spyratos 81], [Hegner 03], the main schema is decomposed into two *meet-complementary* views.
- One is isomorphic to the view schema and tracks its updates exactly.
- The other is held constant for all updates to the view.
- It can be shown [Hegner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

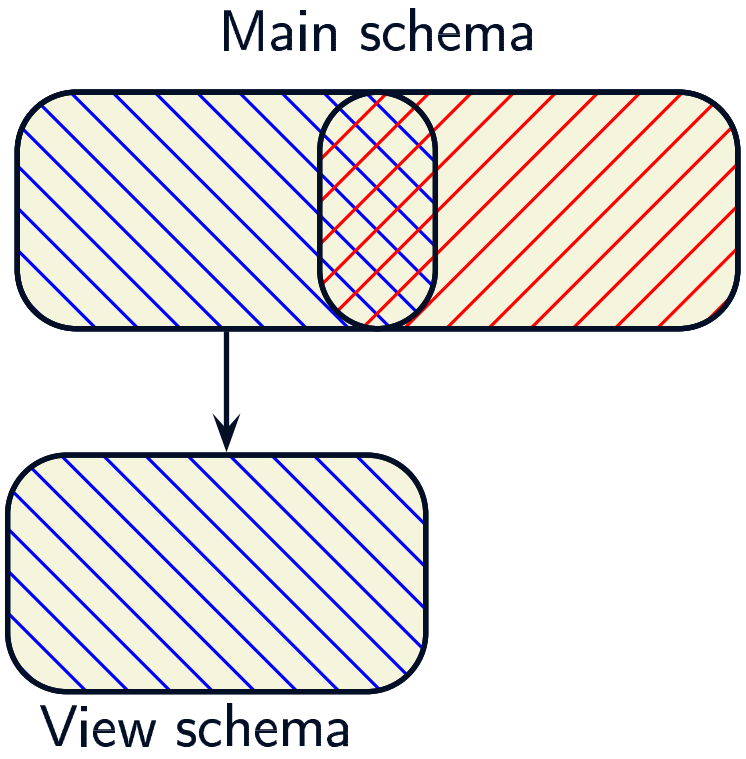



- Consequently, it is quite limited in the view updates which it allows.

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

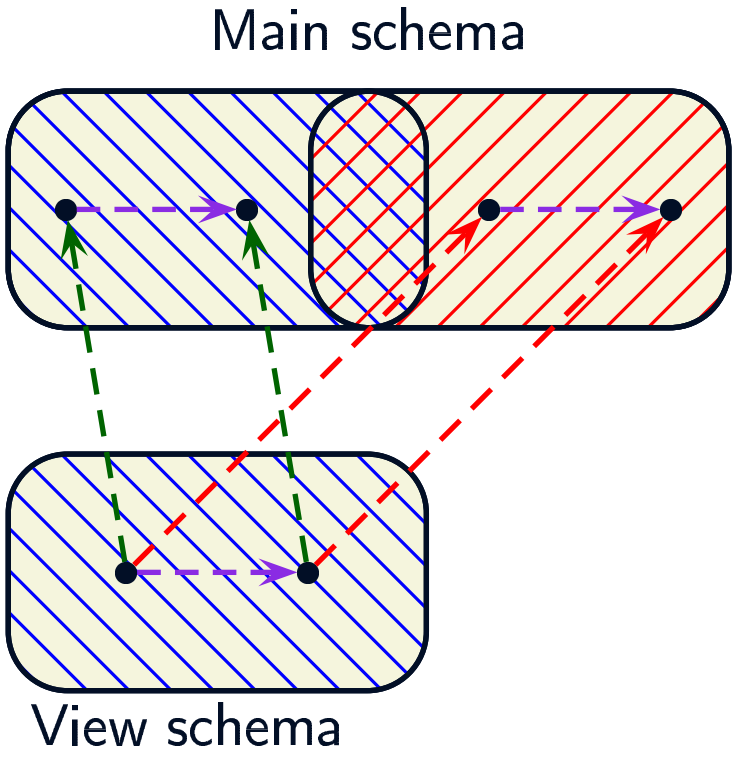
# Moving Beyond the Constant-Complement Strategy

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



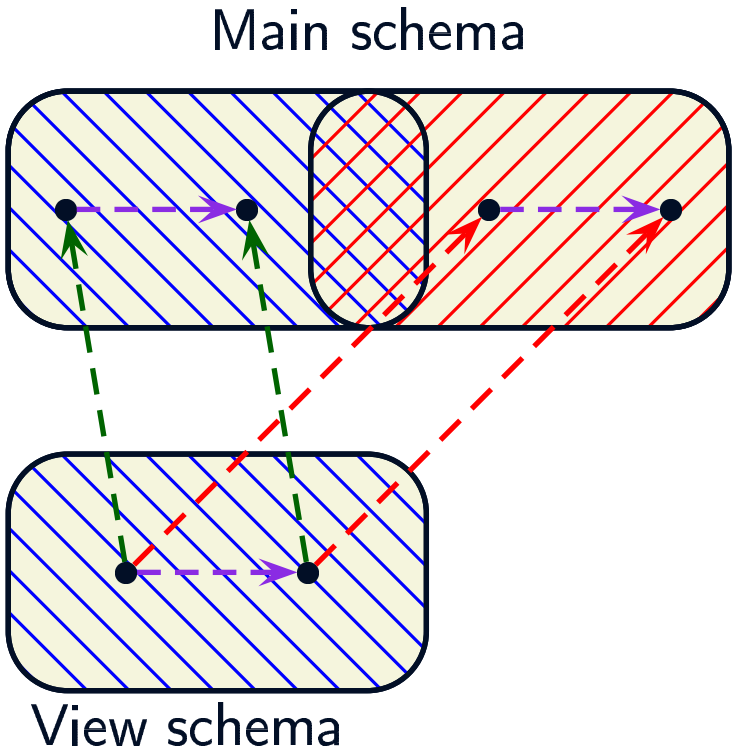
# Moving Beyond the Constant-Complement Strategy

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



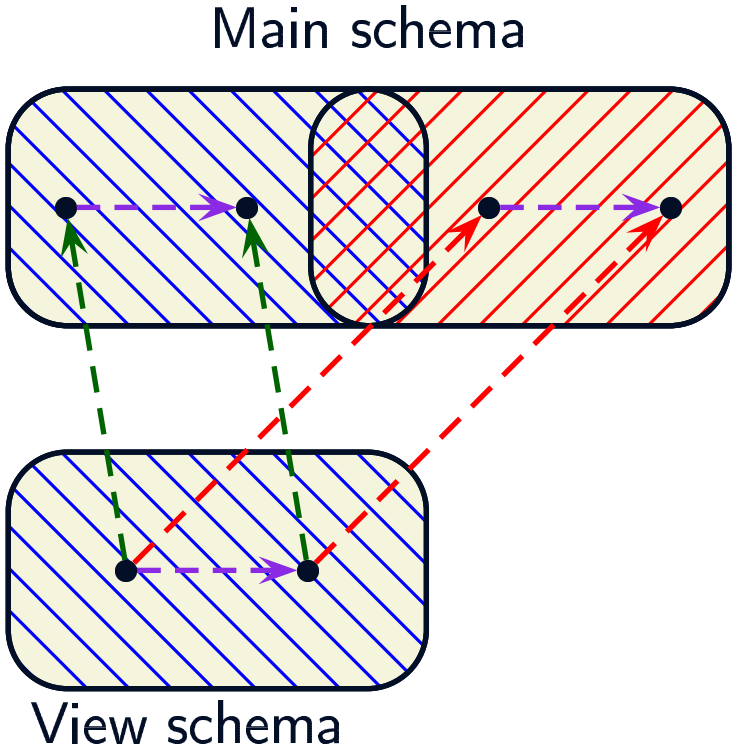
# Moving Beyond the Constant-Complement Strategy

- Over the years, many extensions to the constant-complement strategy have been proposed, all with 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.



# Moving Beyond the Constant-Complement Strategy

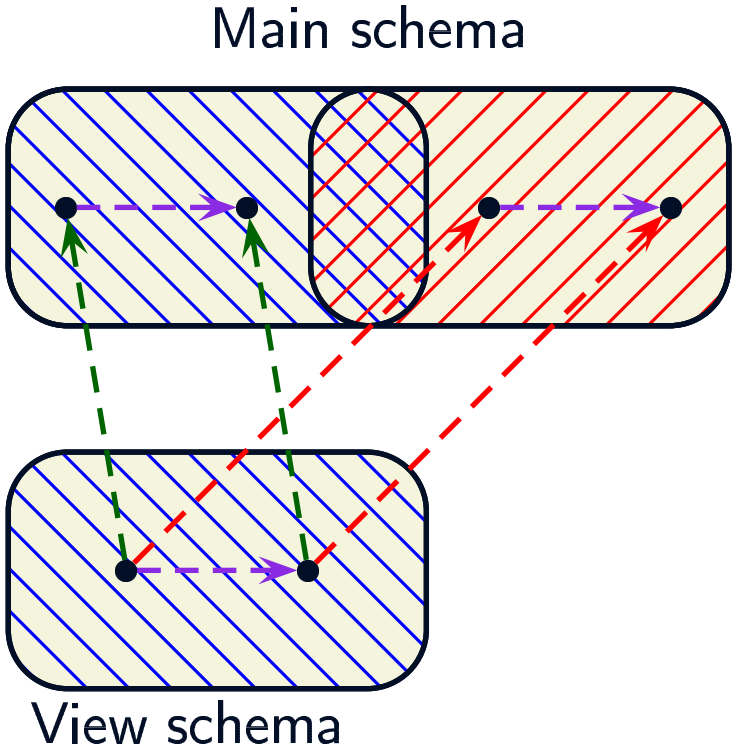
- Over the years, many extensions to the constant-complement strategy have been proposed, all with the following problems.
- Visibility problem: Part of the reflected update is not visible within the view.
- Permission problem: The user of the view lacks the necessary access privileges to effect the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

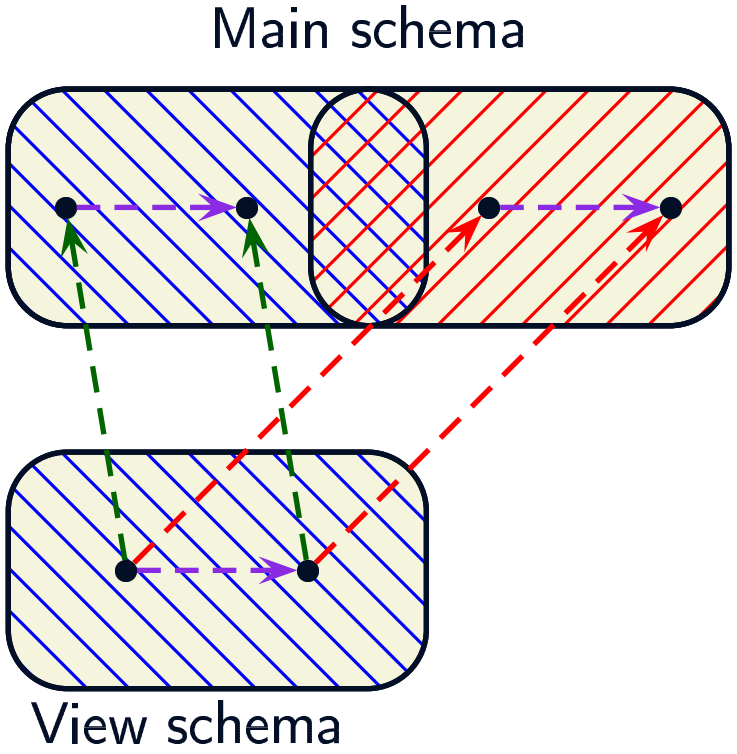



Proposed Solution: *Update by cooperation*

- The user of the view enlists the cooperation of other users to manage 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 with 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 manage 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 Component Model of Database Schemata

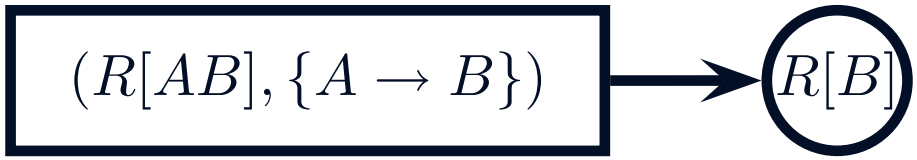
- The idea of modelling a large database schema as the interconnection of smaller *database components* has been forwarded recently by Thalheim [DKE2005].

# The Component Model of Database Schemata

- The idea of modelling a large database schema as the interconnection of smaller *database components* has been forwarded recently by Thalheim [DKE2005].
- The model employed here is due to Hegner [EJC07], and is based upon *communicating views*, illustrated by a simple example below.

# The Component Model of Database Schemata

- The idea of modelling a large database schema as the interconnection of smaller *database components* has been forwarded recently by Thalheim [DKE2005].
- The model employed here is due to Hegner [EJC07], and is based upon *communicating views*, illustrated by a simple example below.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$

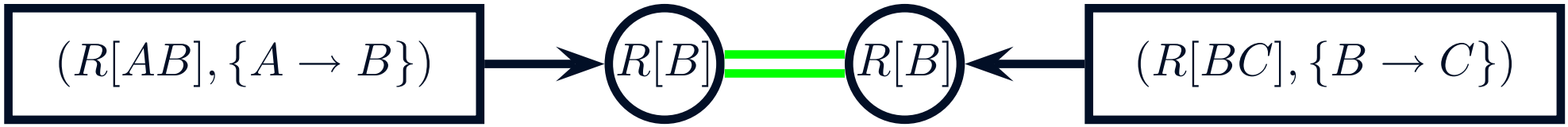


# The Component Model of Database Schemata

- The idea of modelling a large database schema as the interconnection of smaller *database components* has been forwarded recently by Thalheim [DKE2005].
- The model employed here is due to Hegner [EJC07], and is based upon *communicating views*, illustrated by a simple example below.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  and $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ .

# The Component Model of Database Schemata

- The idea of modelling a large database schema as the interconnection of smaller *database components* has been forwarded recently by Thalheim [DKE2005].
- The model employed here is due to Hegner [EJC07], and is based upon *communicating views*, illustrated by a simple example below.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  and $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ .
- Connecting the ports of these two components results in a combination which is isomorphic to $(R[ABC], \{A \to B, B \to C\})$.

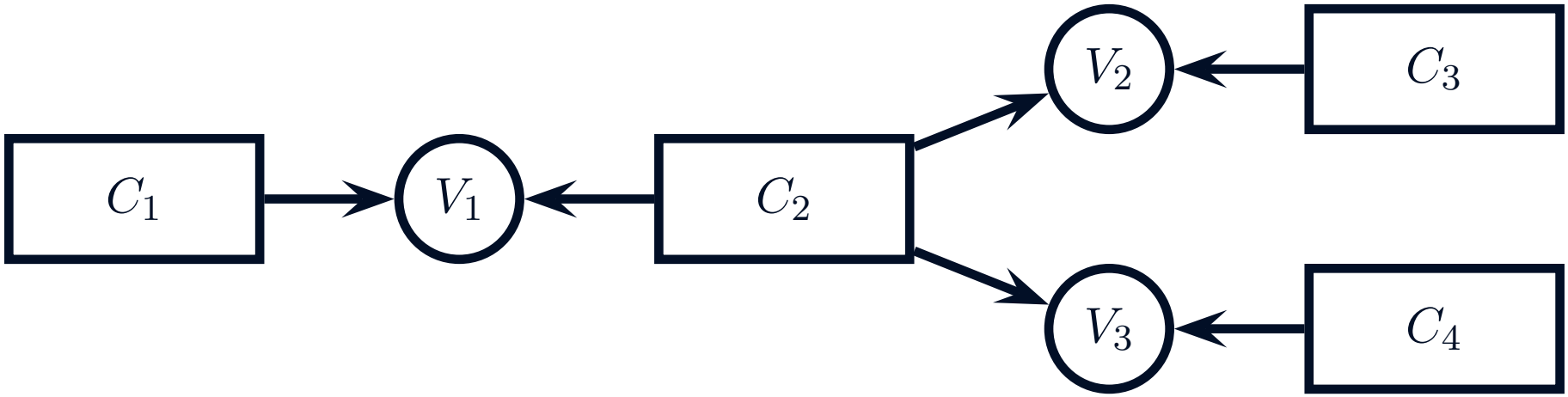


# The Component Model of Database Schemata

- The idea of modelling a large database schema as the interconnection of smaller *database components* has been forwarded recently by Thalheim [DKE2005].
- The model employed here is due to Hegner [EJC07], and is based upon *communicating views*, illustrated by a simple example below.
- Define the component $K_{AB} = ((R[AB], \{A \to B\}), \{\Pi_B^{R[AB]}\})$
  and $K_{BC} = ((R[BC], \{B \to C\}), \{\Pi_B^{R[BC]}\})$ .
- Connecting the ports of these two components results in a combination which is isomorphic to $(R[ABC], \{A \to B, B \to C\})$.

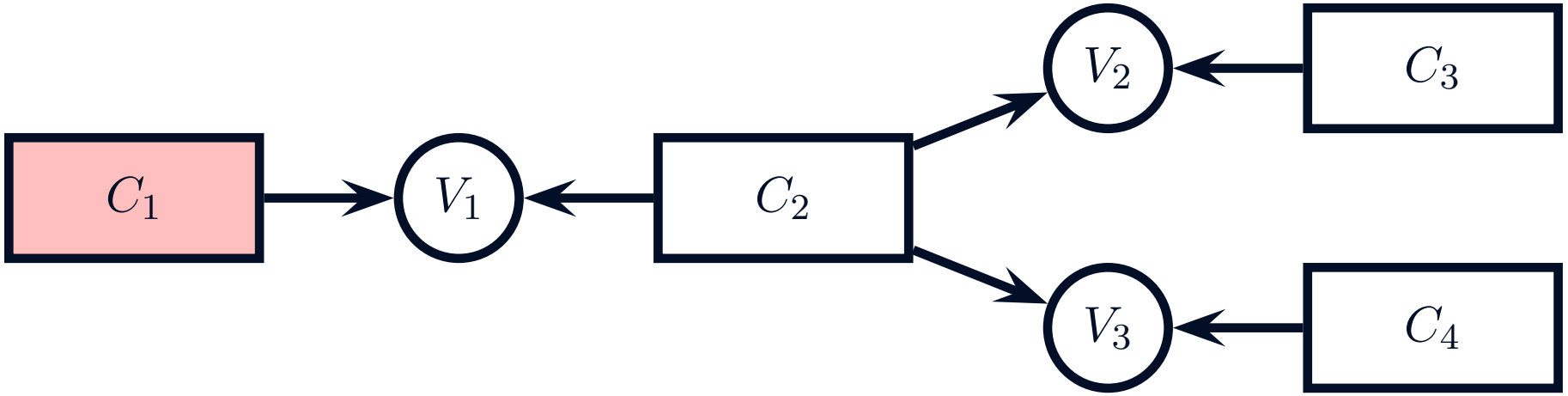

- This recaptures lossless and dependency-preserving decomposition, but as a *composition* rather than as a decomposition.

# The Propagation of Updates through Components



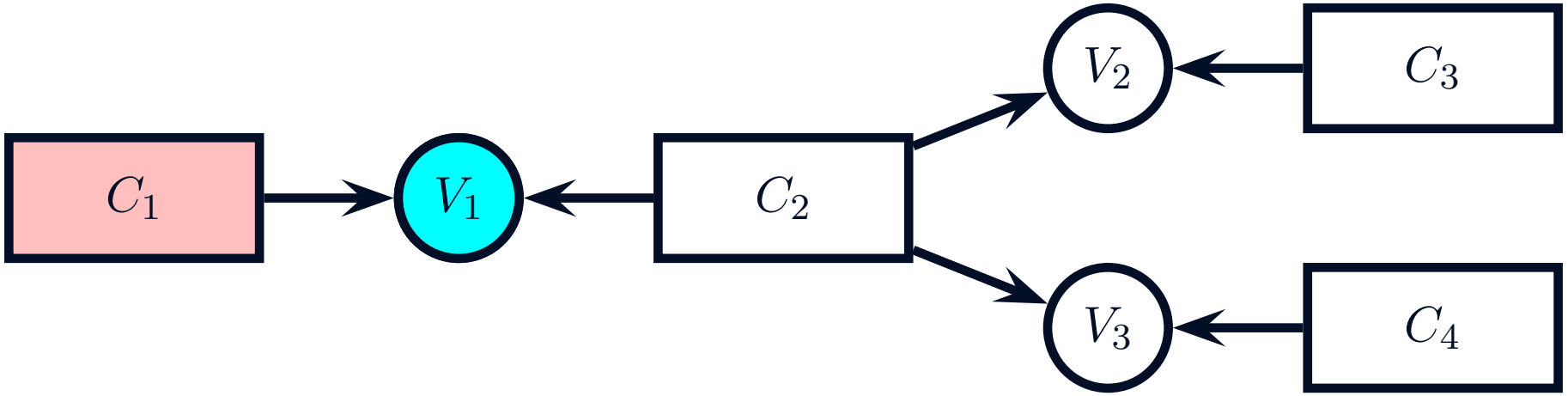
# The Propagation of Updates through Components

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



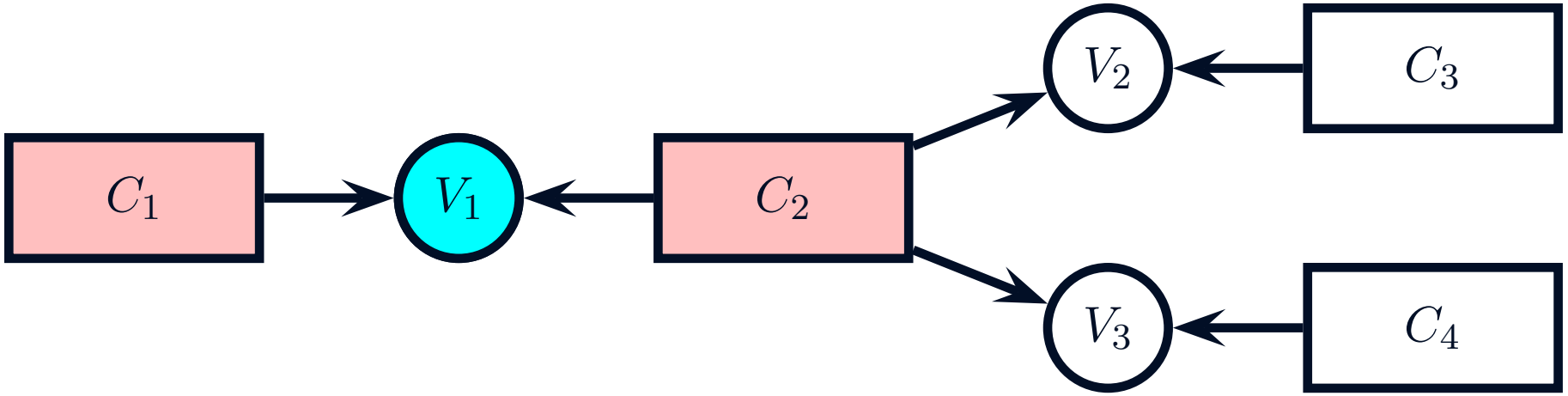
# The Propagation of Updates through Components

- 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.



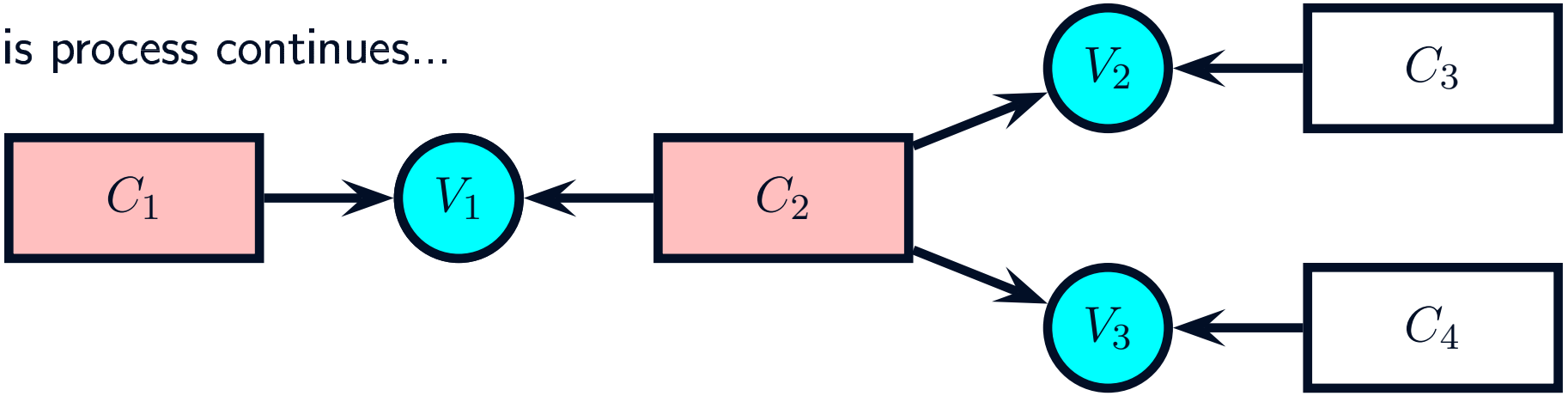
# The Propagation of Updates through Components

- 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 $C_2$.



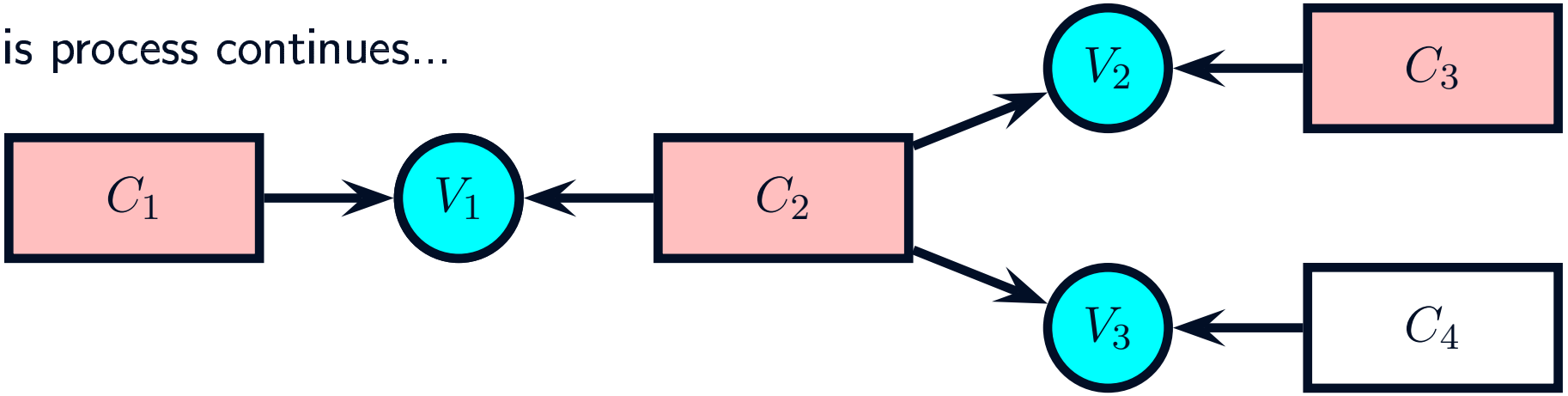
# The Propagation of Updates through Components

- 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...



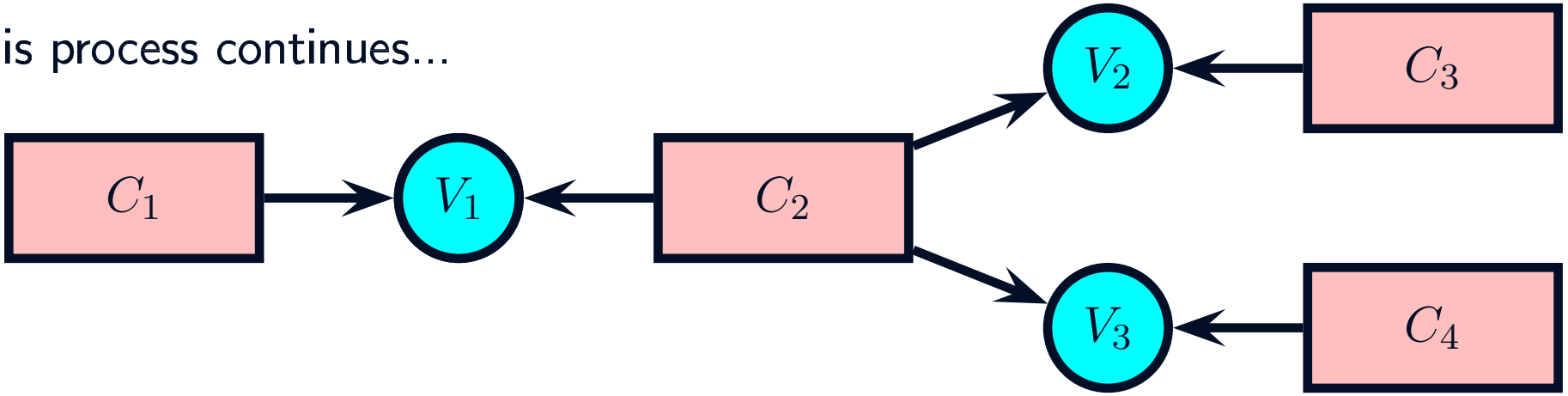
# The Propagation of Updates through Components

- 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...



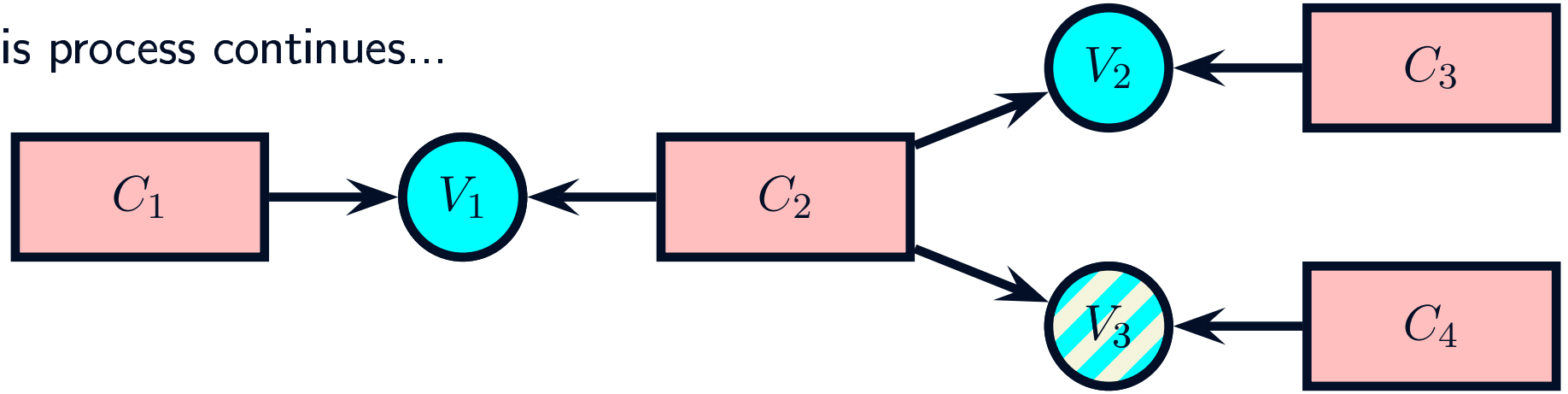
# The Propagation of Updates through Components

- 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...



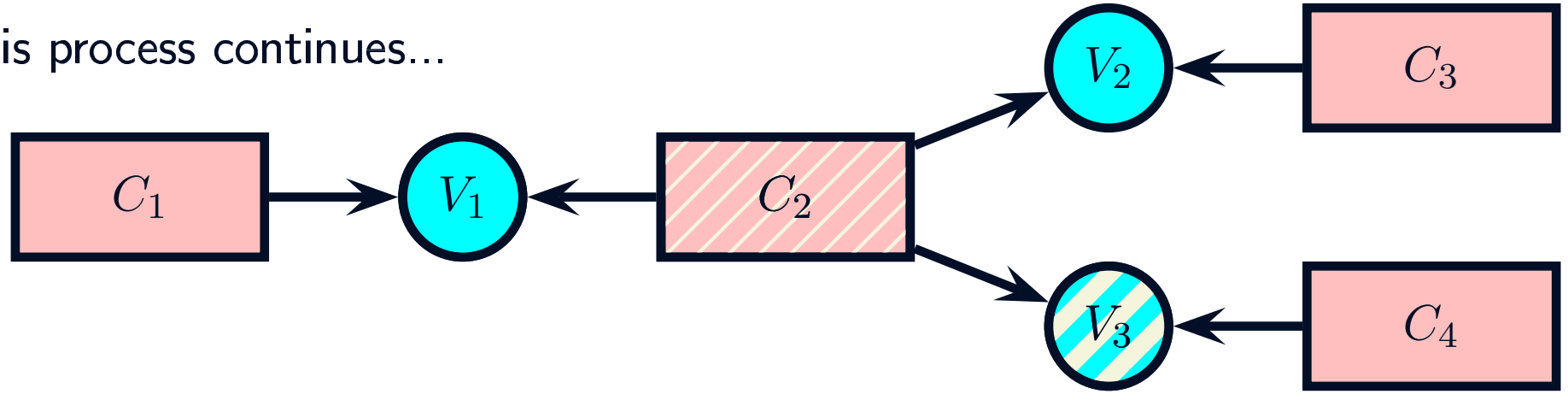
# The Propagation of Updates through Components

- 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...



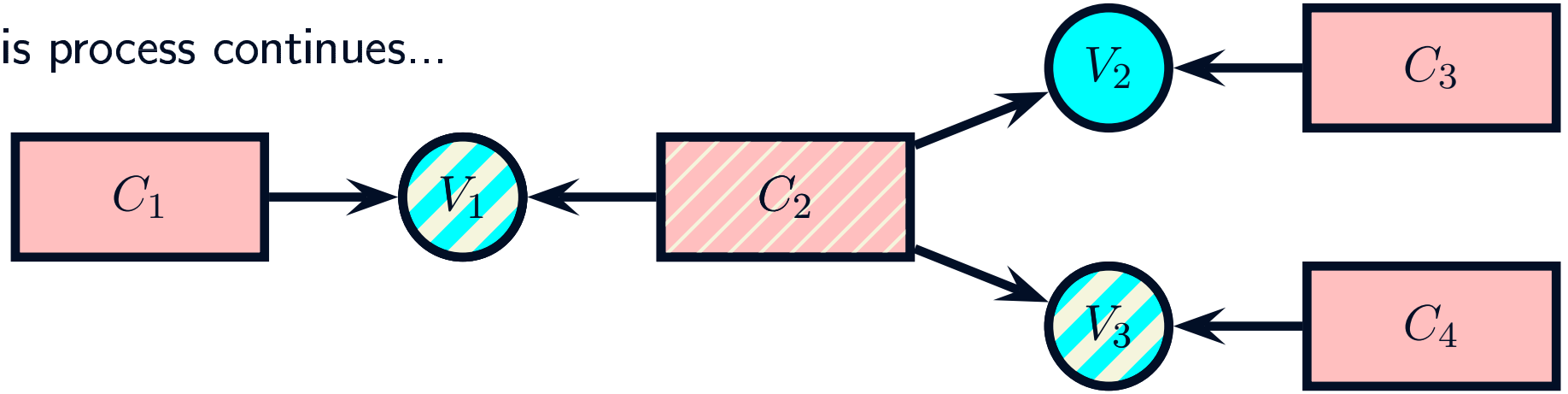
# The Propagation of Updates through Components

- 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...



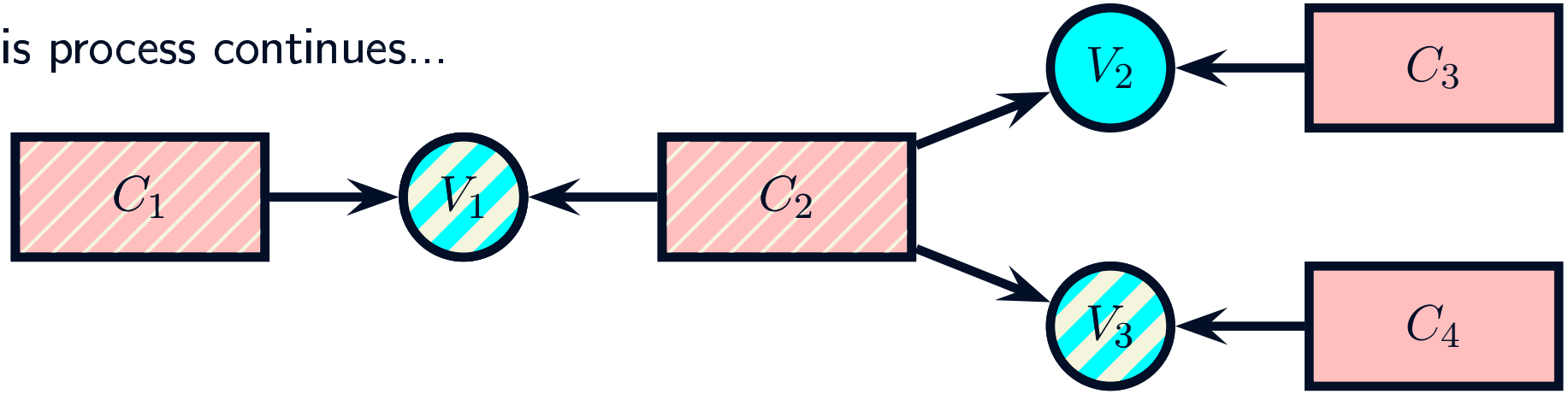
# The Propagation of Updates through Components

- 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...



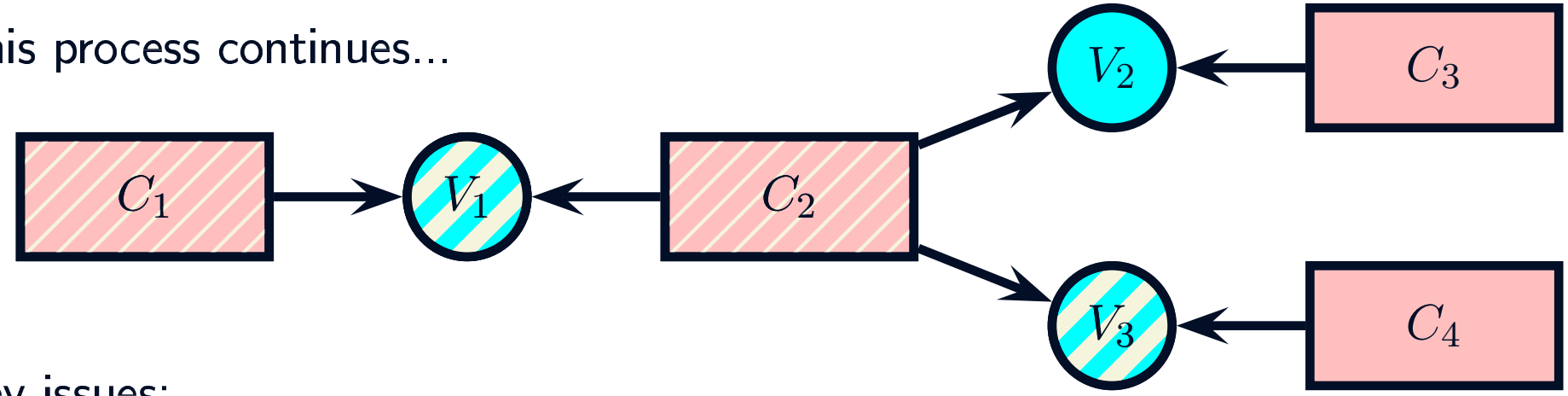
# The Propagation of Updates through Components

- 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

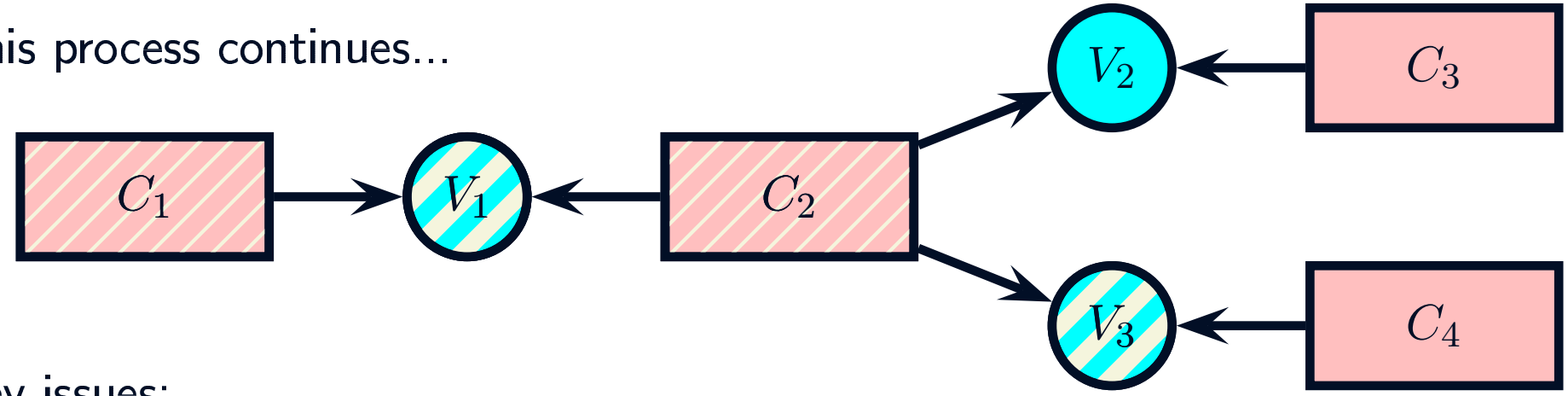
- Suppose that an update to the schema of component $C_1$ is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to $C_2$.
- This process continues...




- Key issues:

# The Propagation of Updates through Components

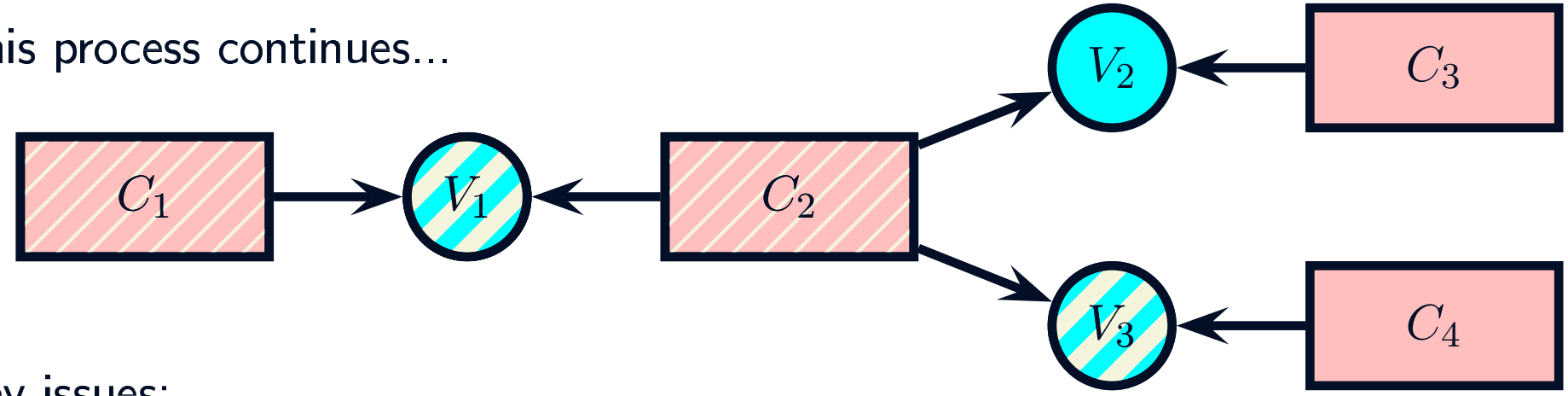
- Suppose that an update to the schema of component $C_1$ is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to $C_2$.
- This process continues...




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

# The Propagation of Updates through Components

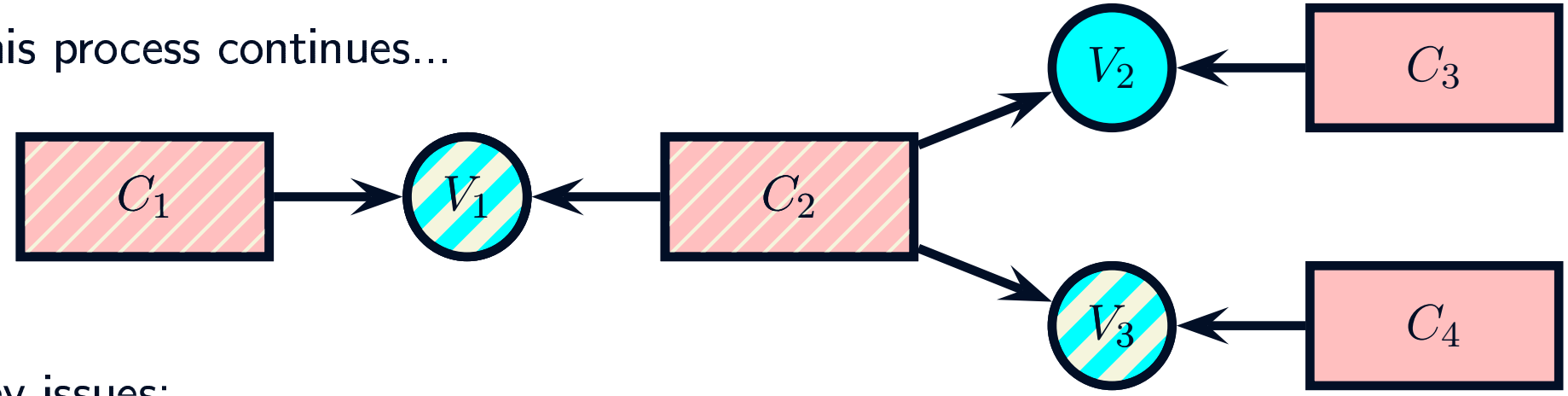
- Suppose that an update to the schema of component $C_1$ is proposed.
- This may require an update to the port schema $\mathbf{V}_1$ as well.
- In turn, this will require a *lifting* of that update to $C_2$.
- This process continues...




- 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.

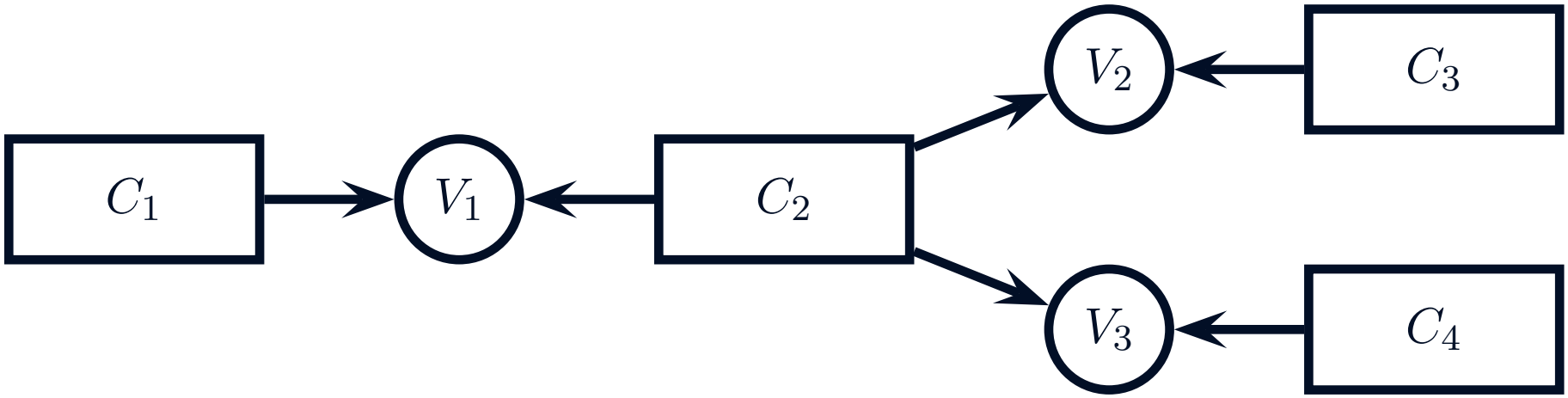
# The Propagation of Updates through Components

- 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...

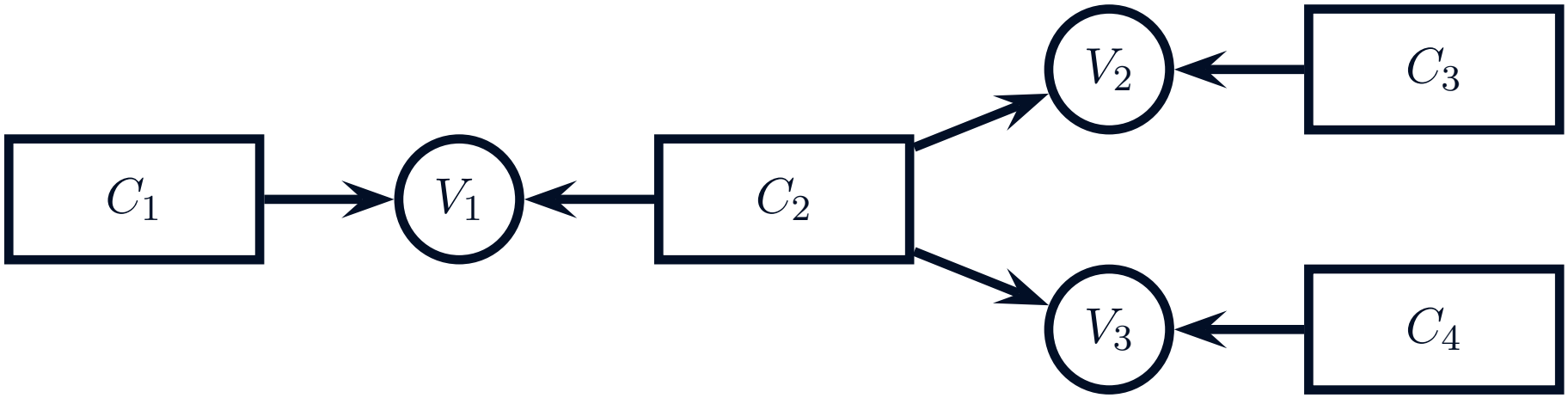



- Key issues:
  - *Database consistency*: Actual database update must be deferred until the negotiation process is complete.
  - *Termination*: The negotiation process must not go on endlessly.
- An architecture for the support of such *cooperative updates* is needed.

# The Architecture of Cooperative Update Management

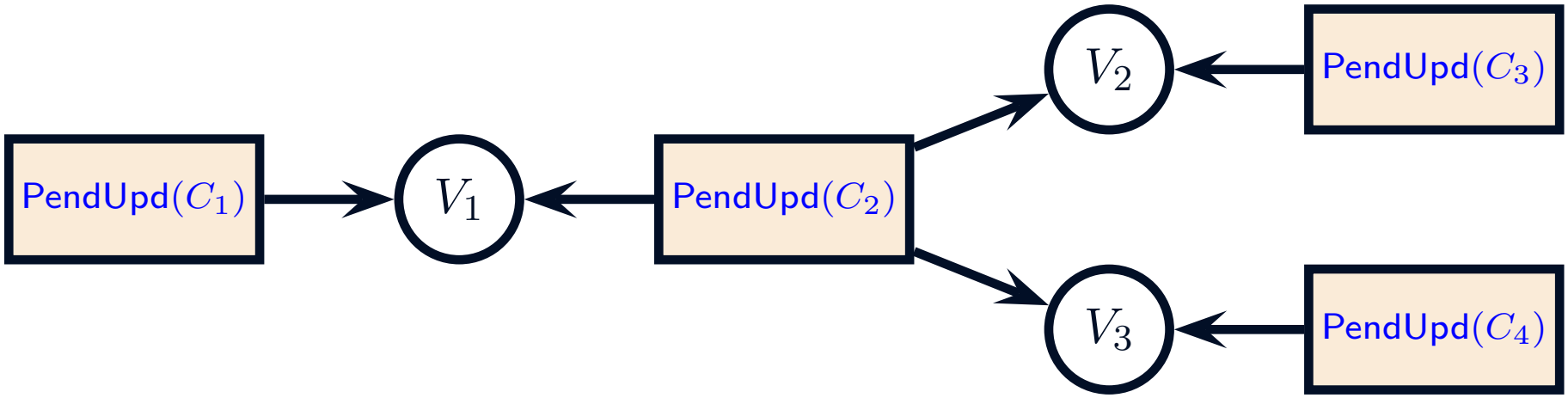


# The Architecture of Cooperative Update Management



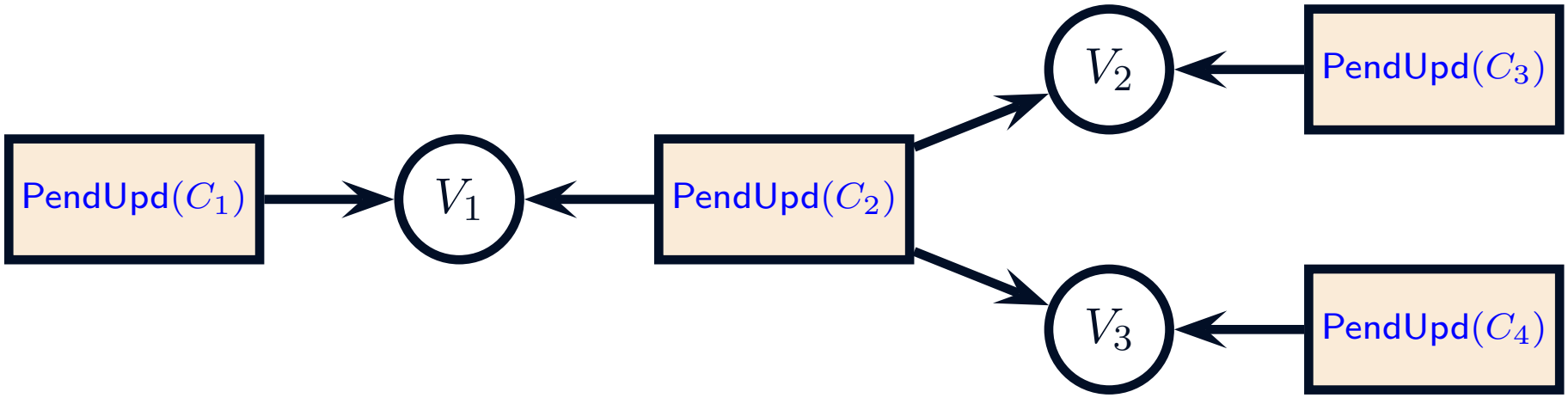

- To each component schema $C_i$ corresponds a *pending update register* PendUpd($C_i$).

# The Architecture of Cooperative Update Management



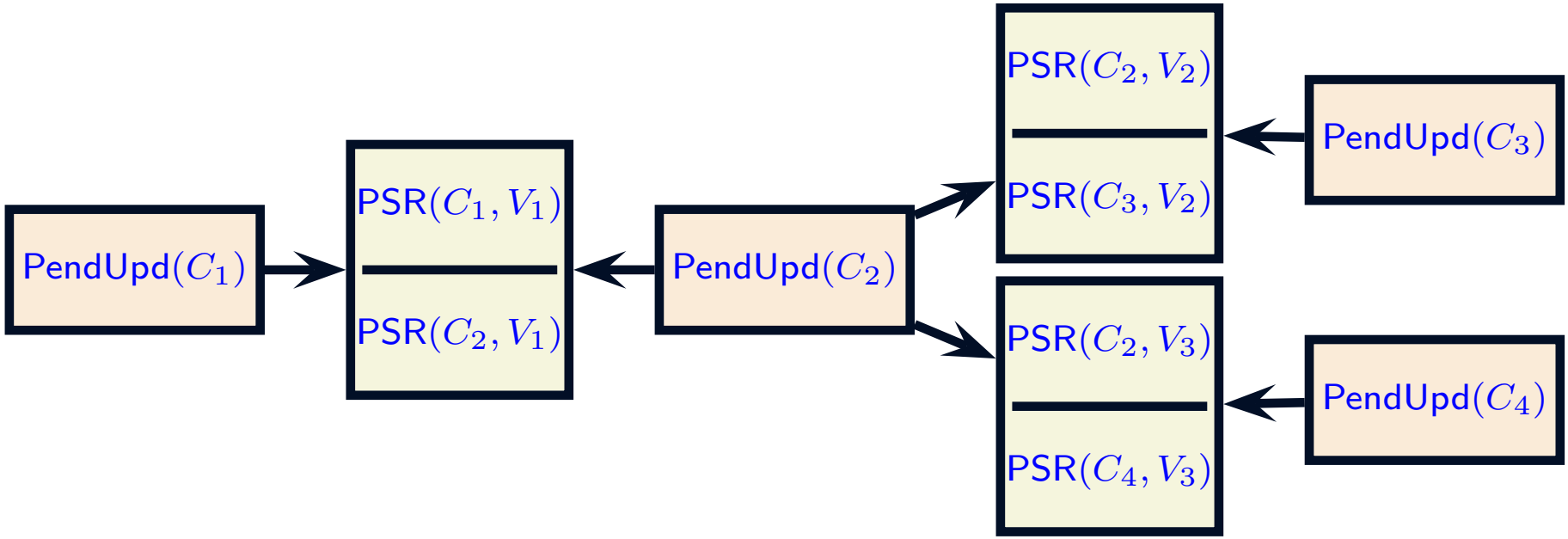

- To each component schema $C_i$ corresponds a *pending update register* PendUpd($C_i$).

# The Architecture of Cooperative Update Management



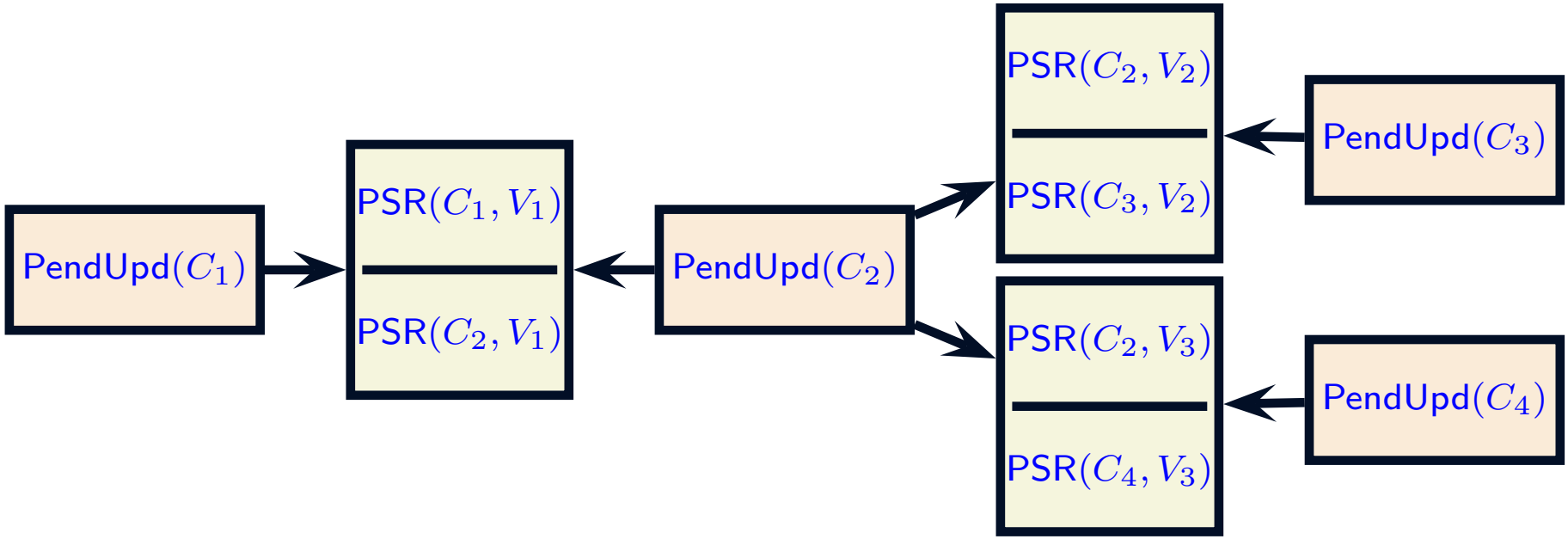

- To each component schema $C_i$ corresponds a *pending update register* PendUpd($C_i$).
- To each view schema $V_i$ is associated a *port status register* PSR($C_j, V_i$) for each component schema $C_j$ which is connected to it.

# The Architecture of Cooperative Update Management



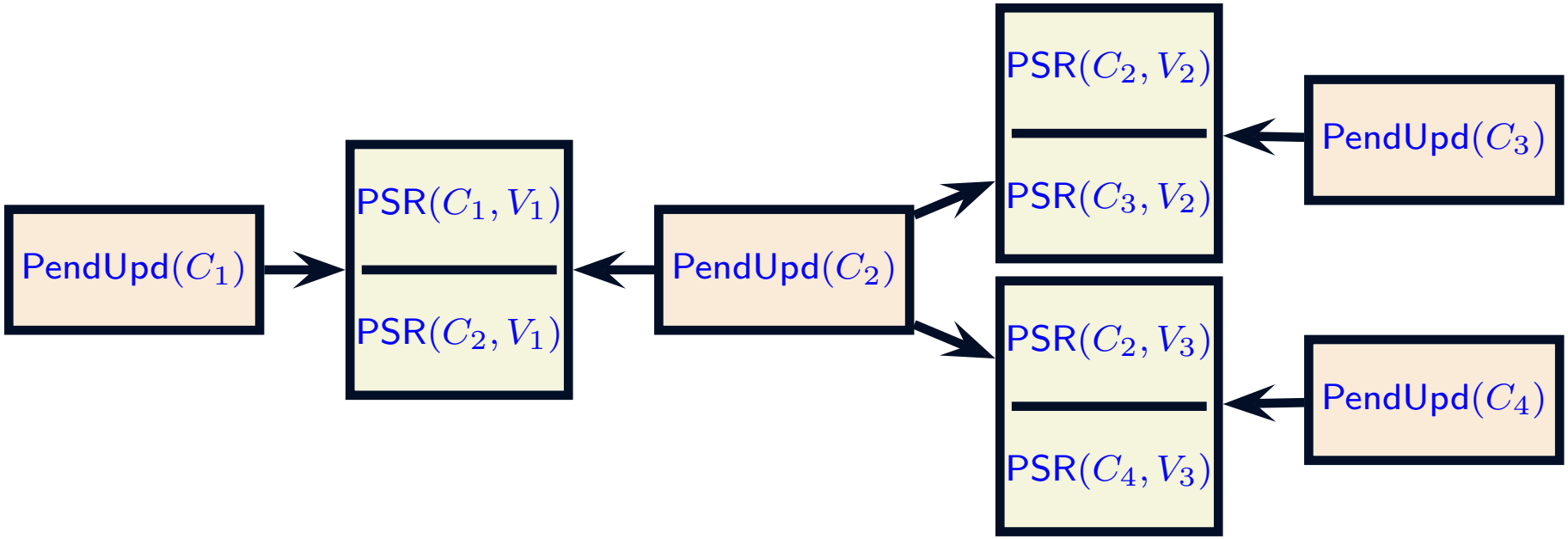

- To each component schema $C_i$ corresponds a *pending update register* PendUpd($C_i$).
- To each view schema $V_i$ is associated a *port status register* PSR($C_j, V_i$) for each component schema $C_j$ which is connected to it.

# The Architecture of Cooperative Update Management



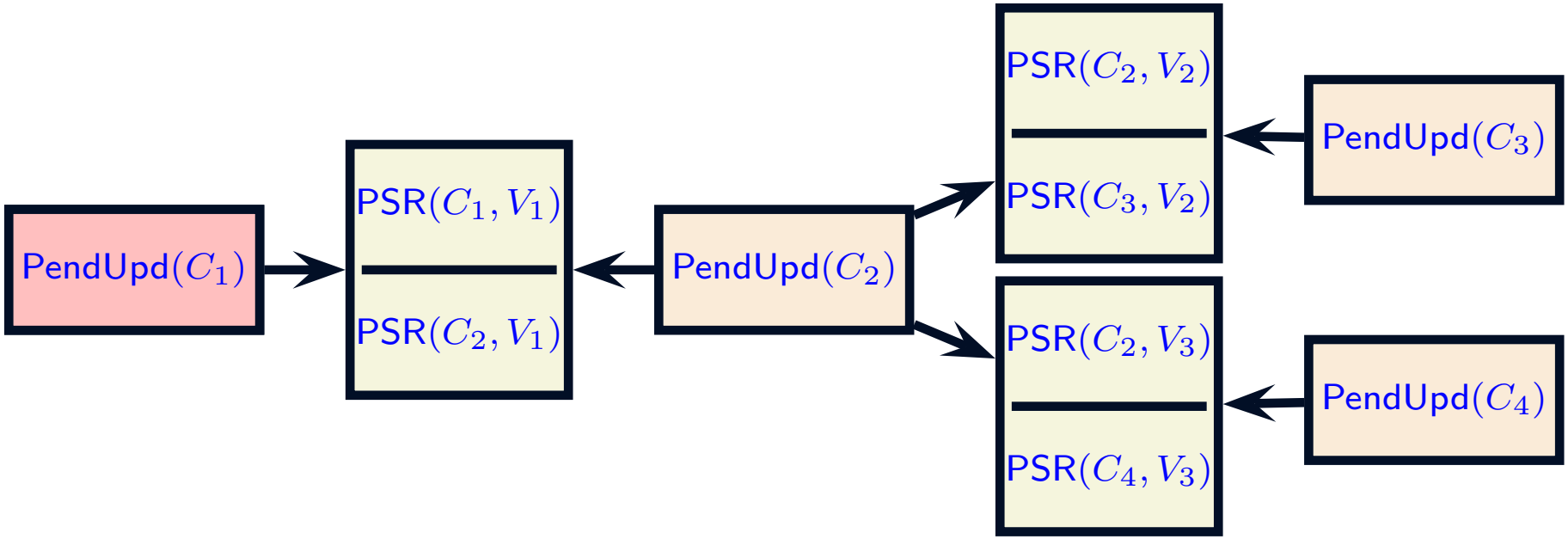

- To each component schema $C_i$ corresponds a *pending update register* PendUpd($C_i$).
- To each view schema $V_i$ is associated a *port status register* PSR($C_j, V_i$) for each component schema $C_j$ which is connected to it.
- These additional registers are part of the control structure, and are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

# Operation of Cooperative Update Management



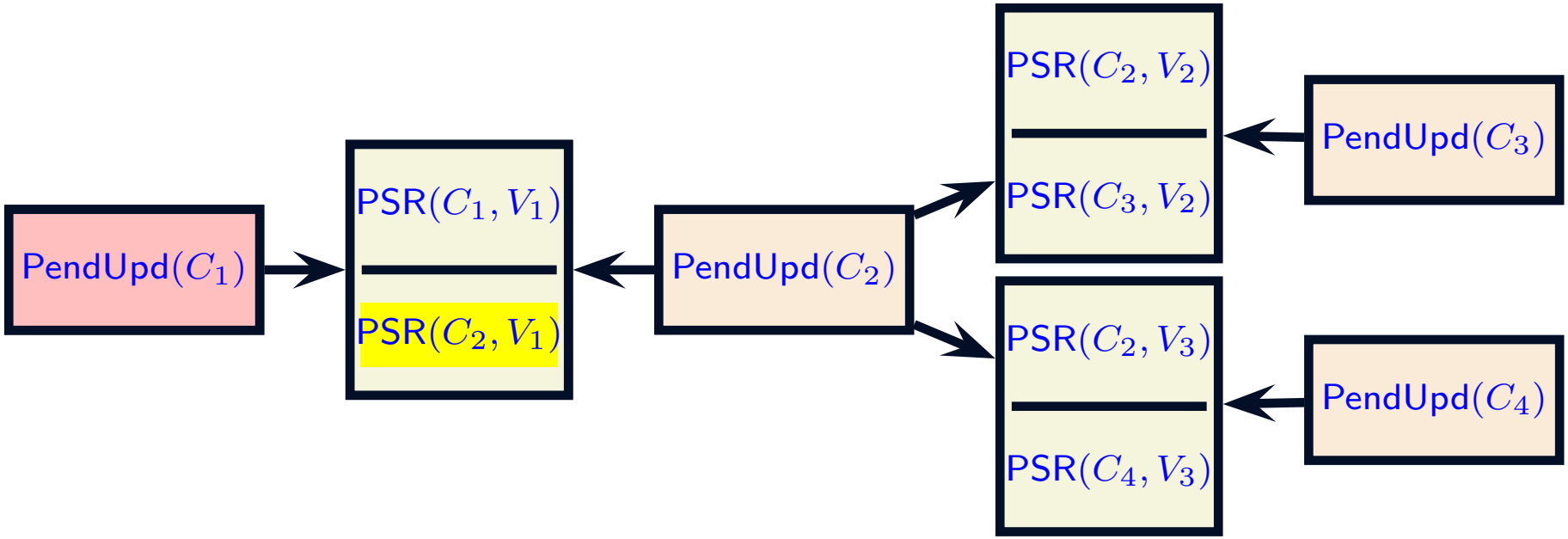

- All port status registers are initially null.

# Operation of Cooperative Update Management



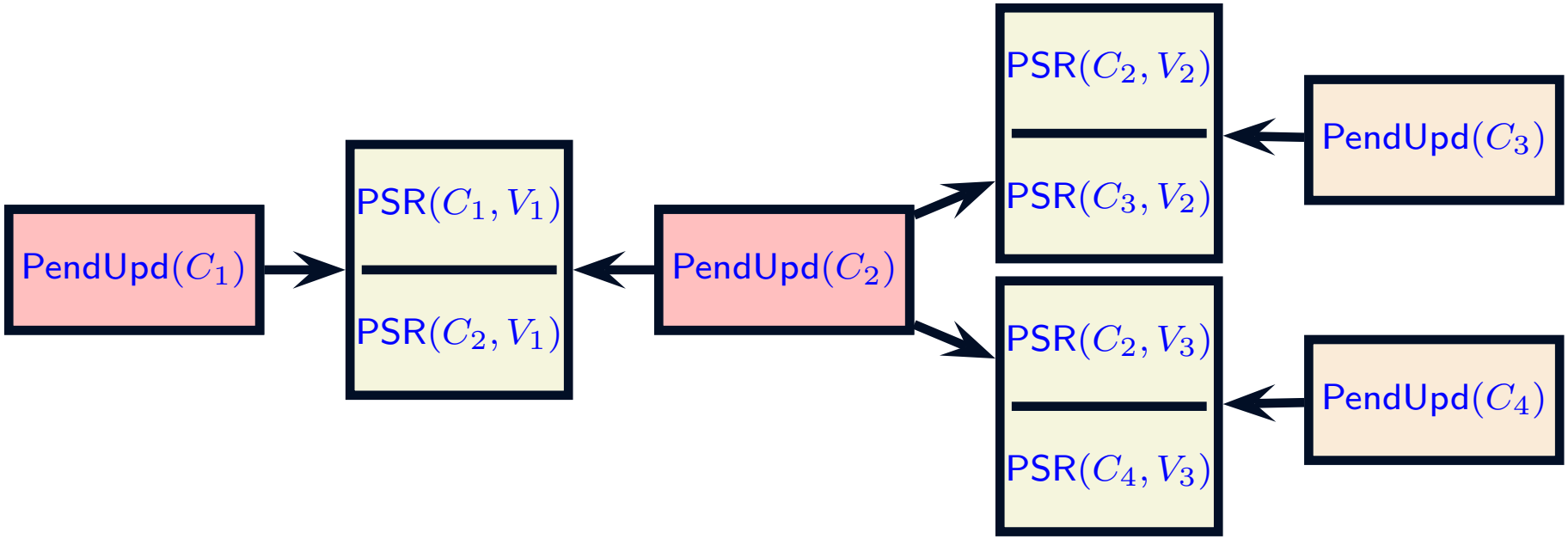

- All port status registers are initially null.
- An update to a component schema

# Operation of Cooperative Update Management



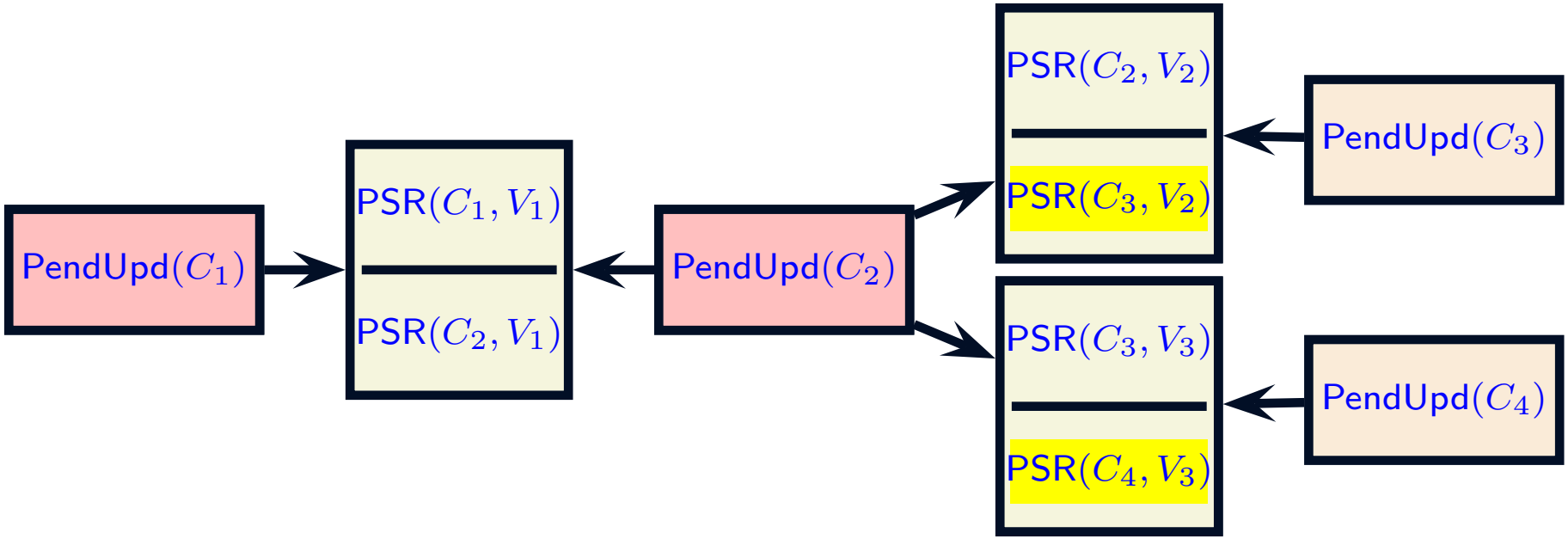

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.

# Operation of Cooperative Update Management



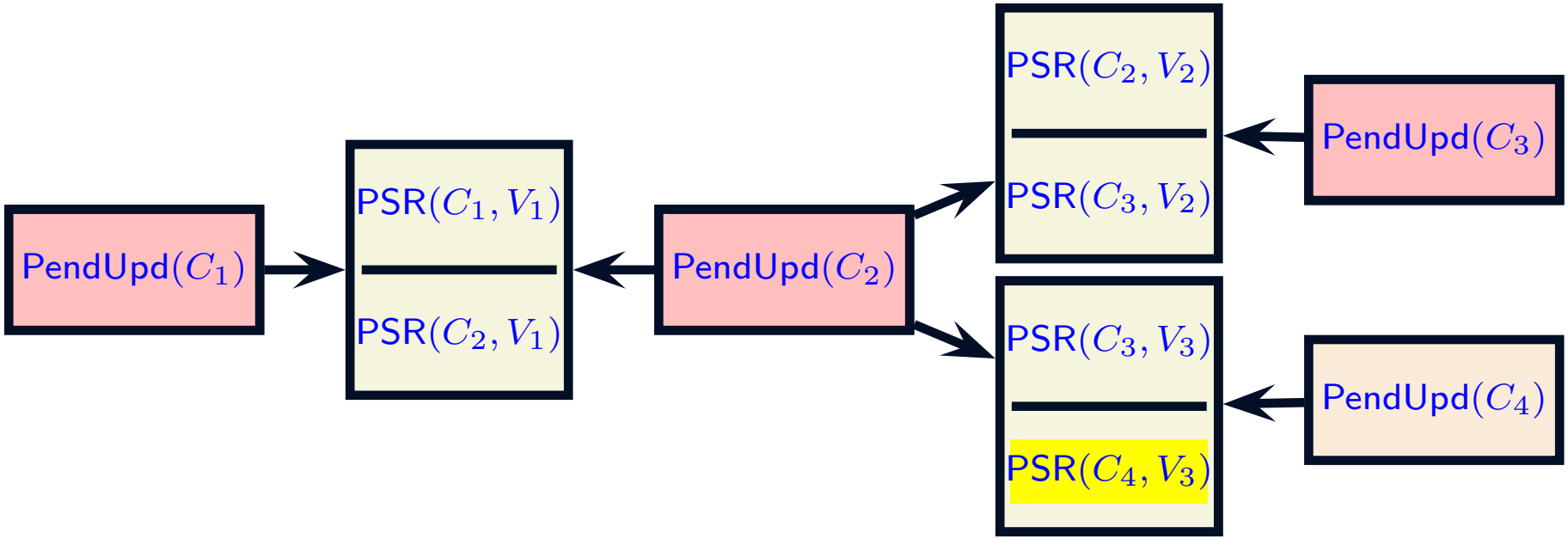

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management



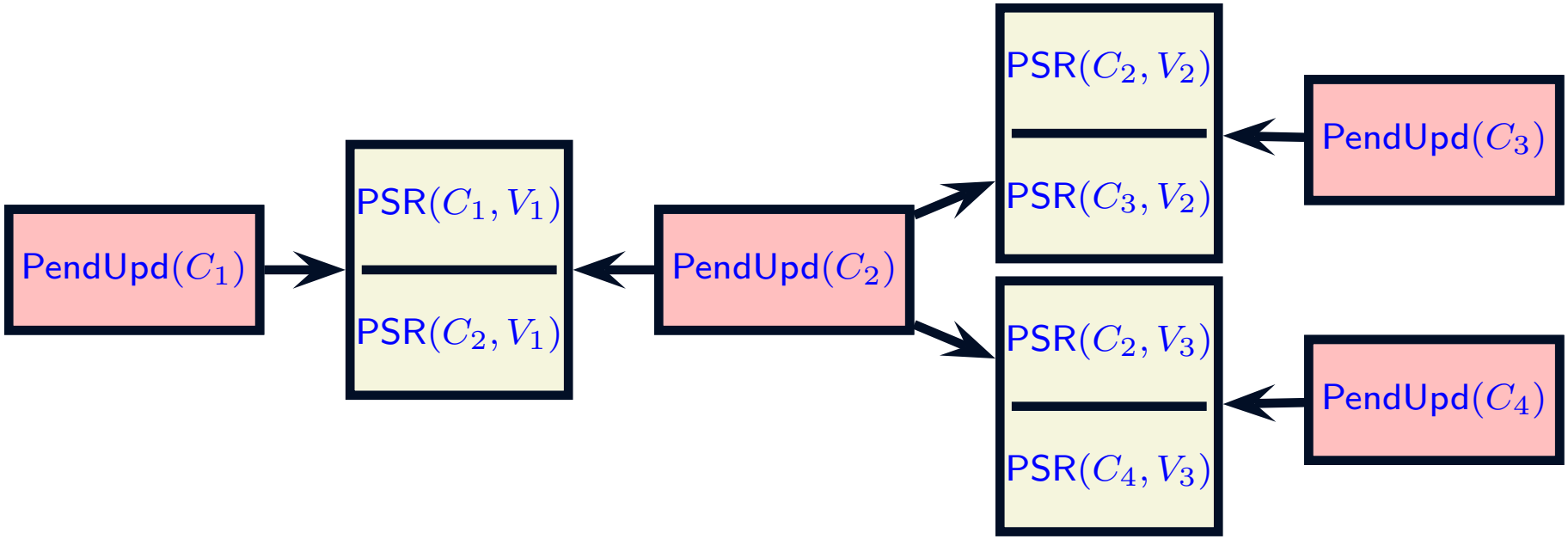

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management



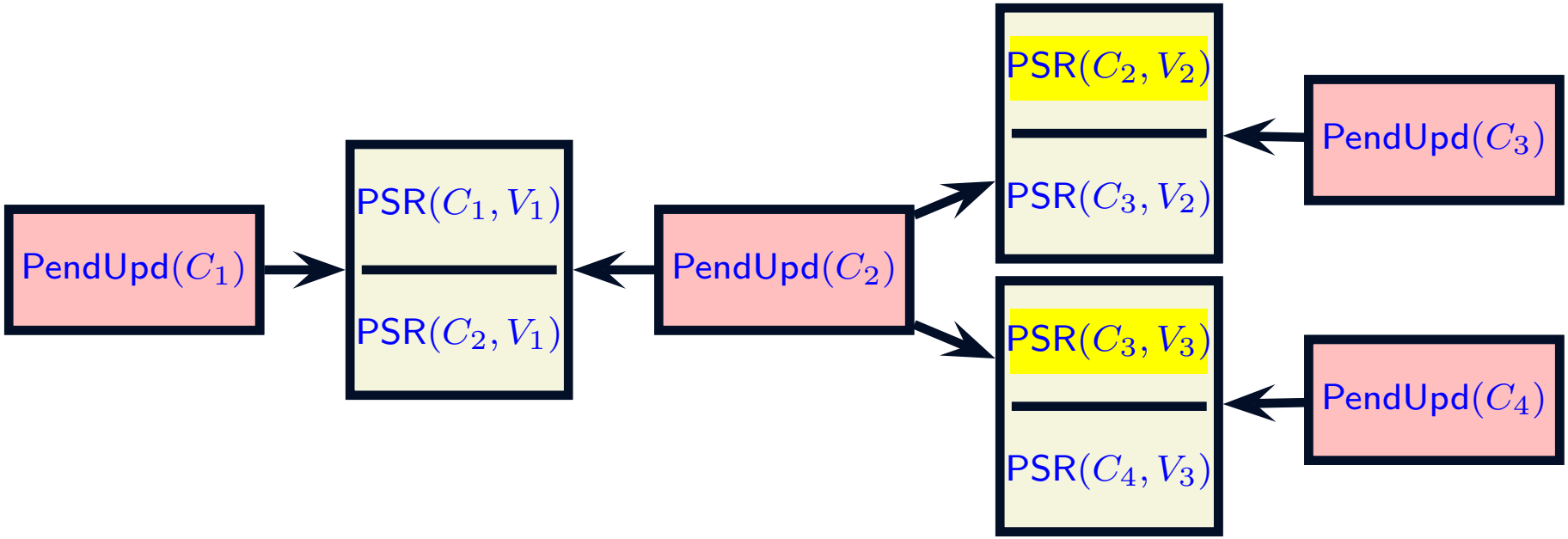

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management



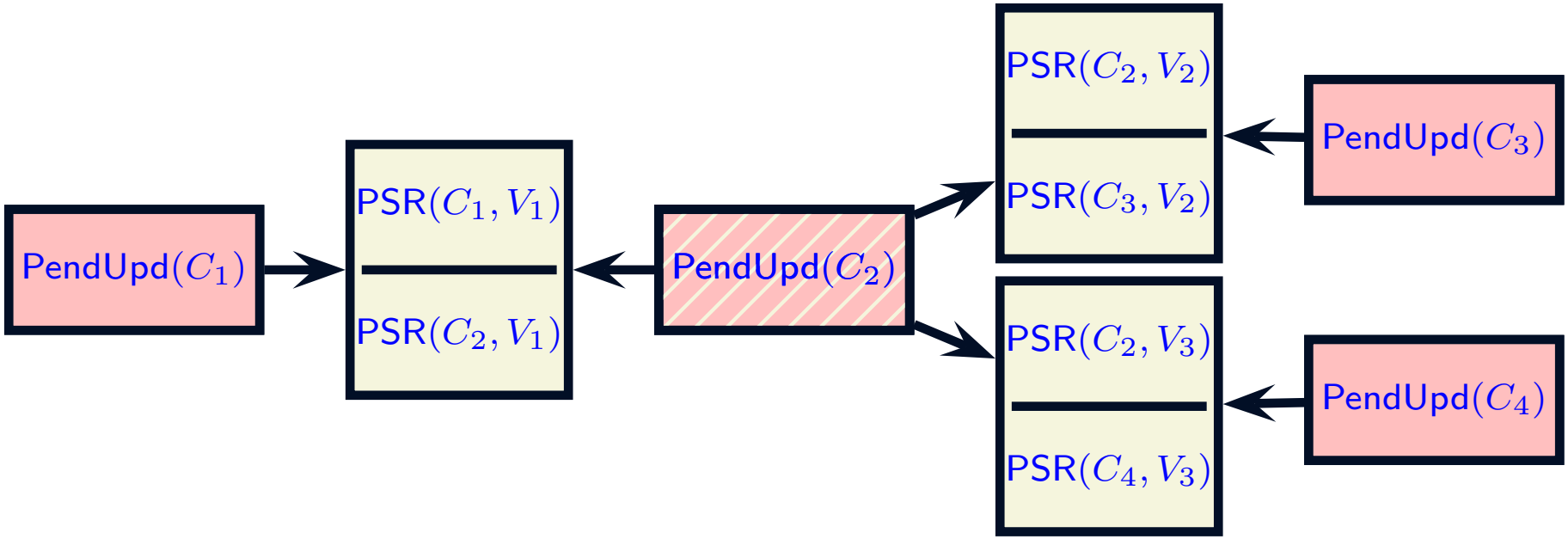

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management



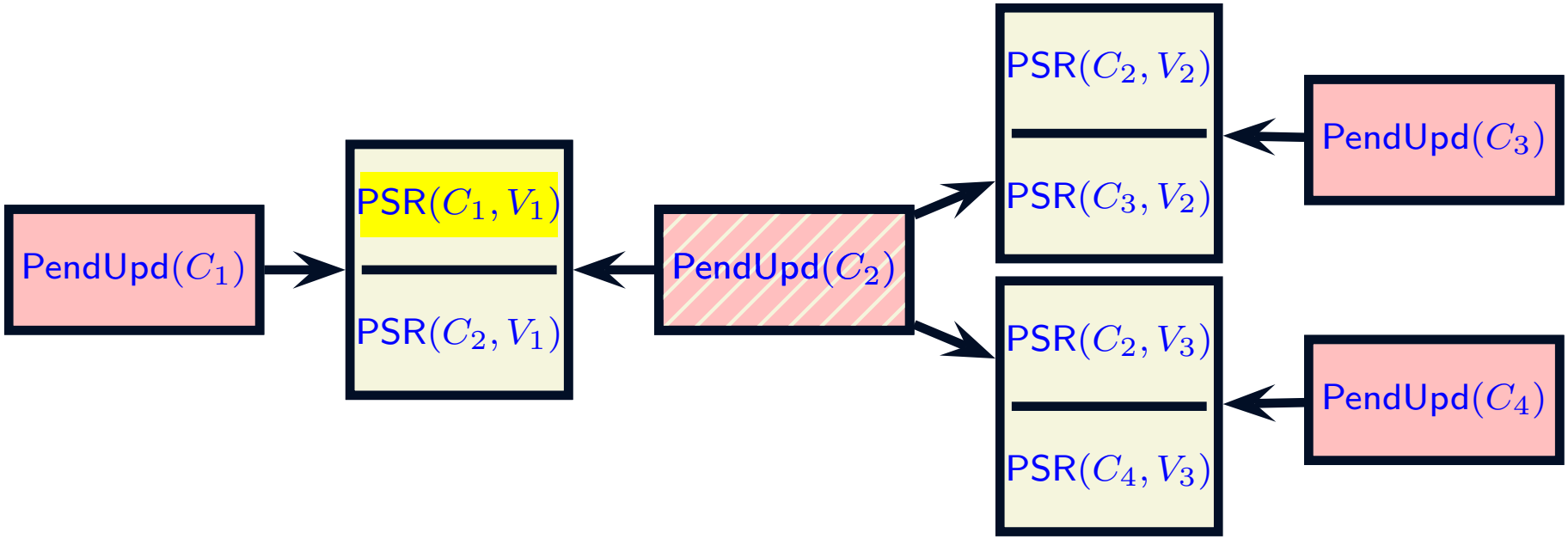

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management



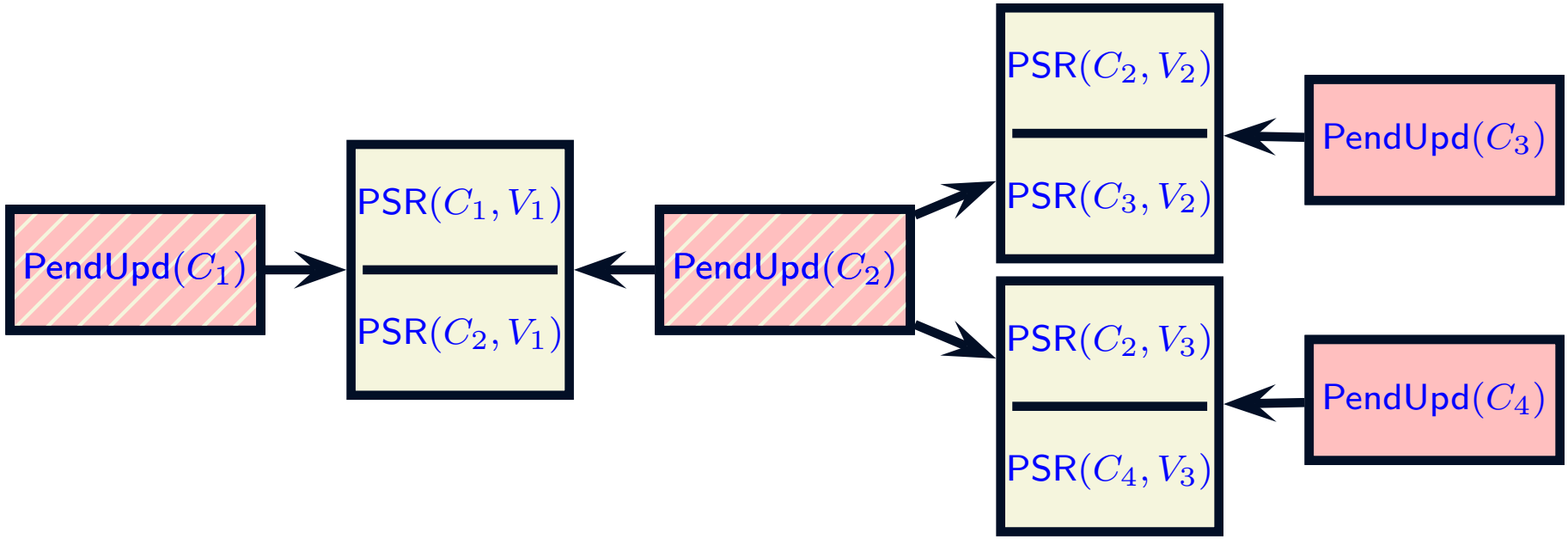

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management



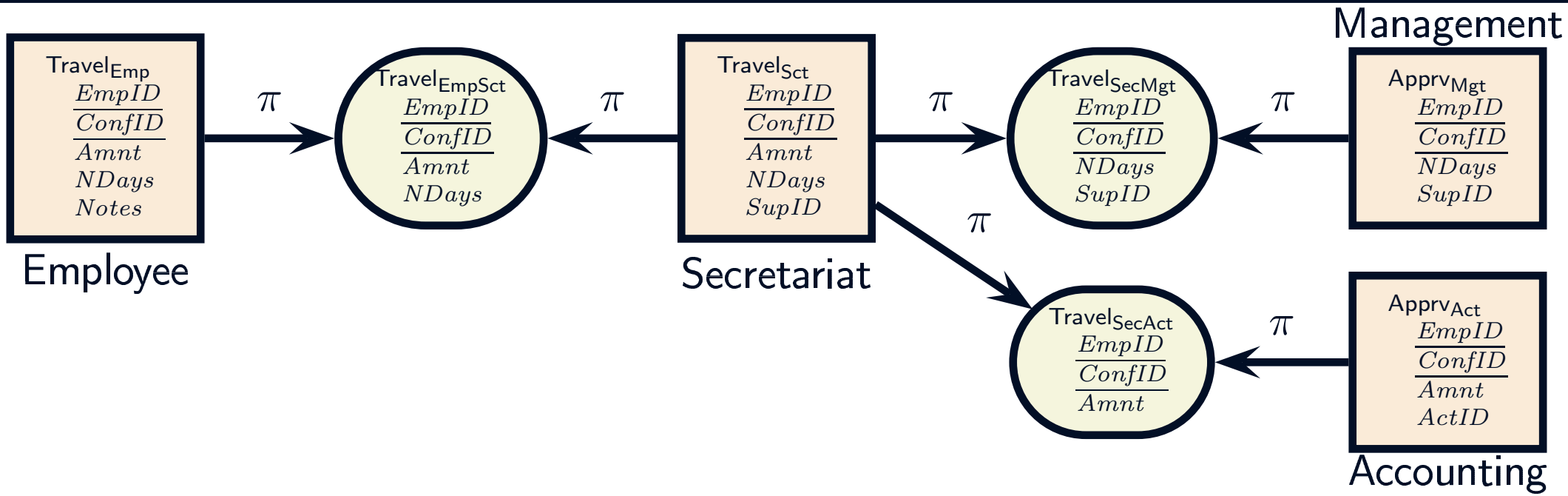

- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Operation of Cooperative Update Management

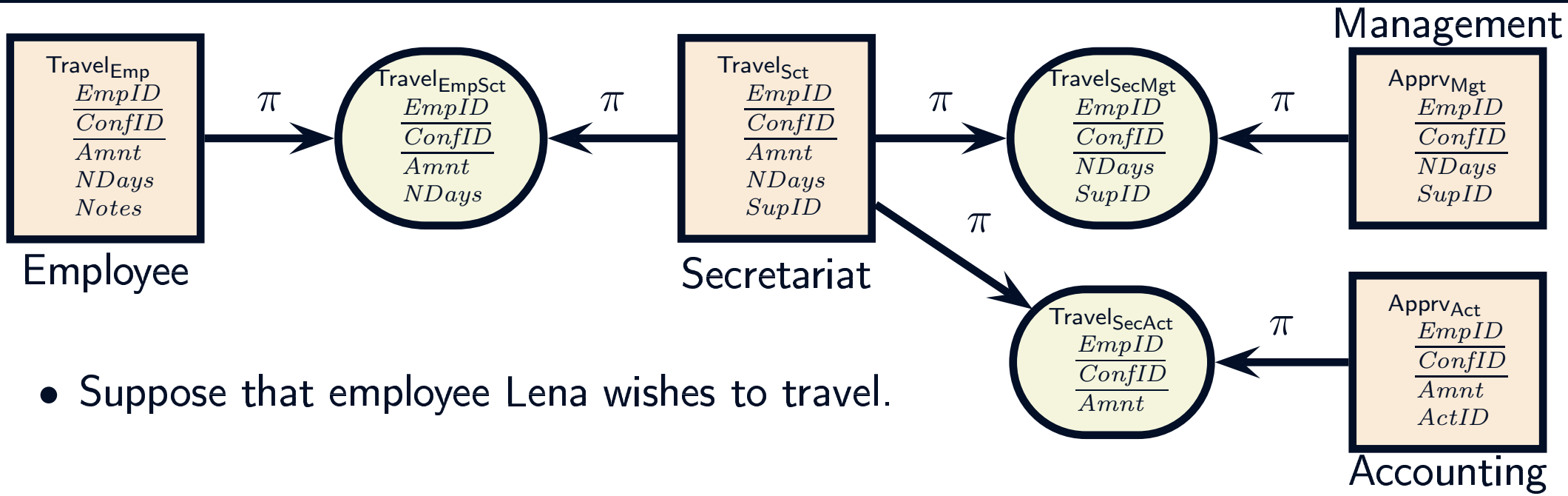



- All port status registers are initially null.
- An update to a component schema places the corresponding projection into the port status registers of its neighbors.
- The neighbor then identifies a suitable *lifting* of the view state in the port-status register to its schema, and resets that port-status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

# Example: Travel Request and Authorization

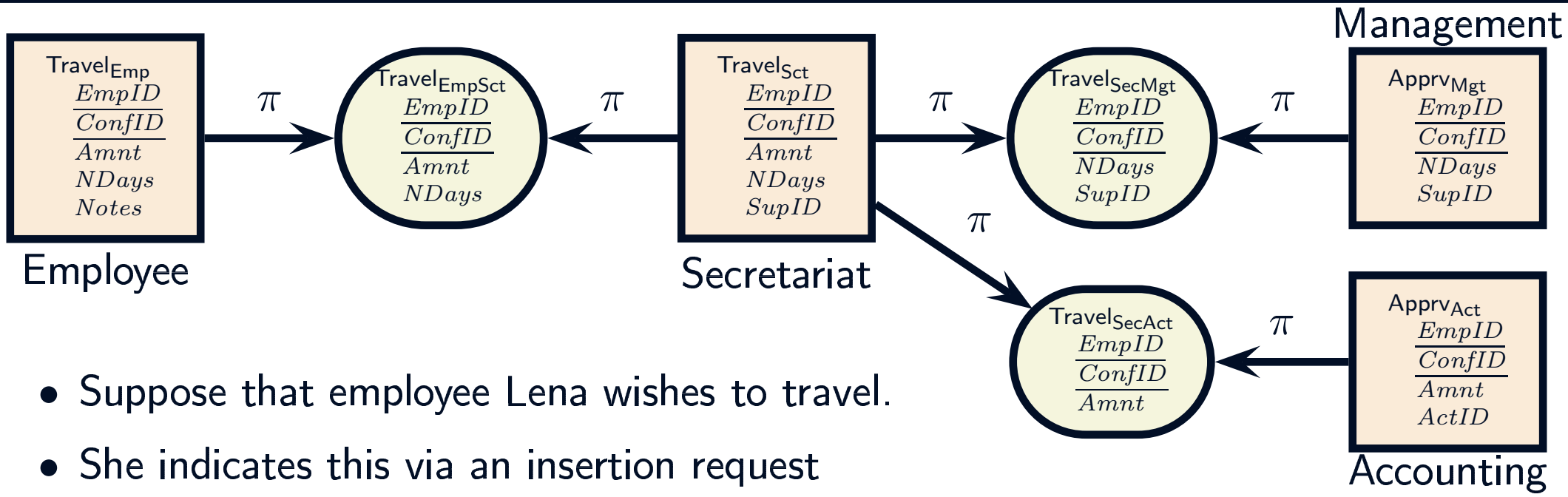


# Example: Travel Request and Authorization



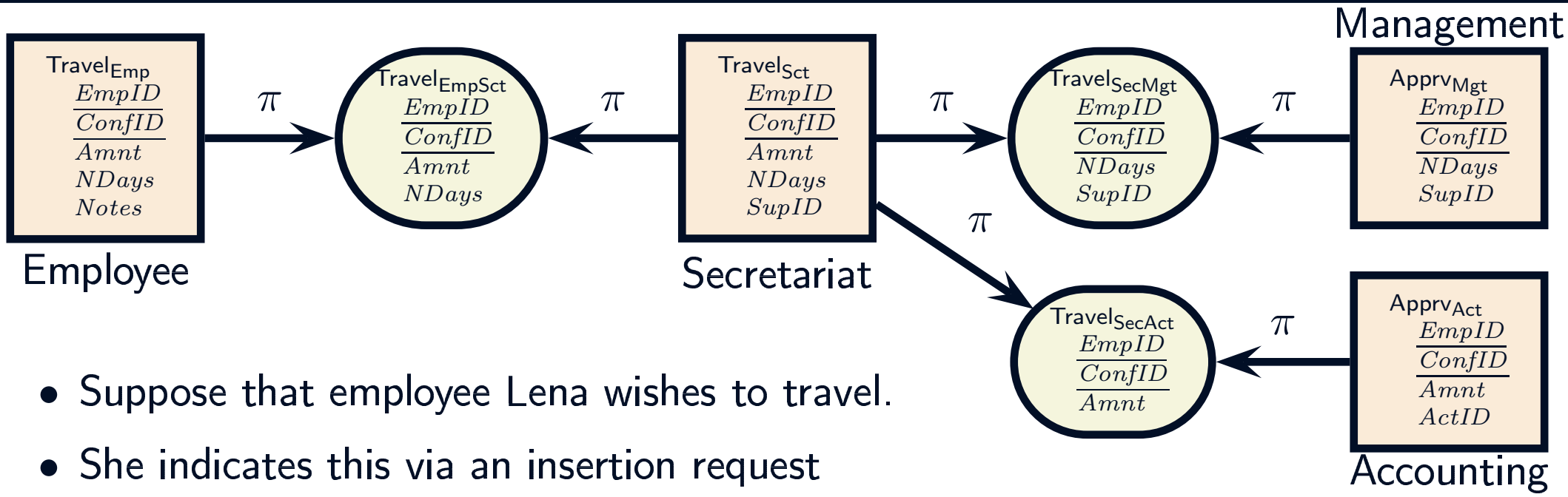

- Suppose that employee Lena wishes to travel.

# Example: Travel Request and Authorization



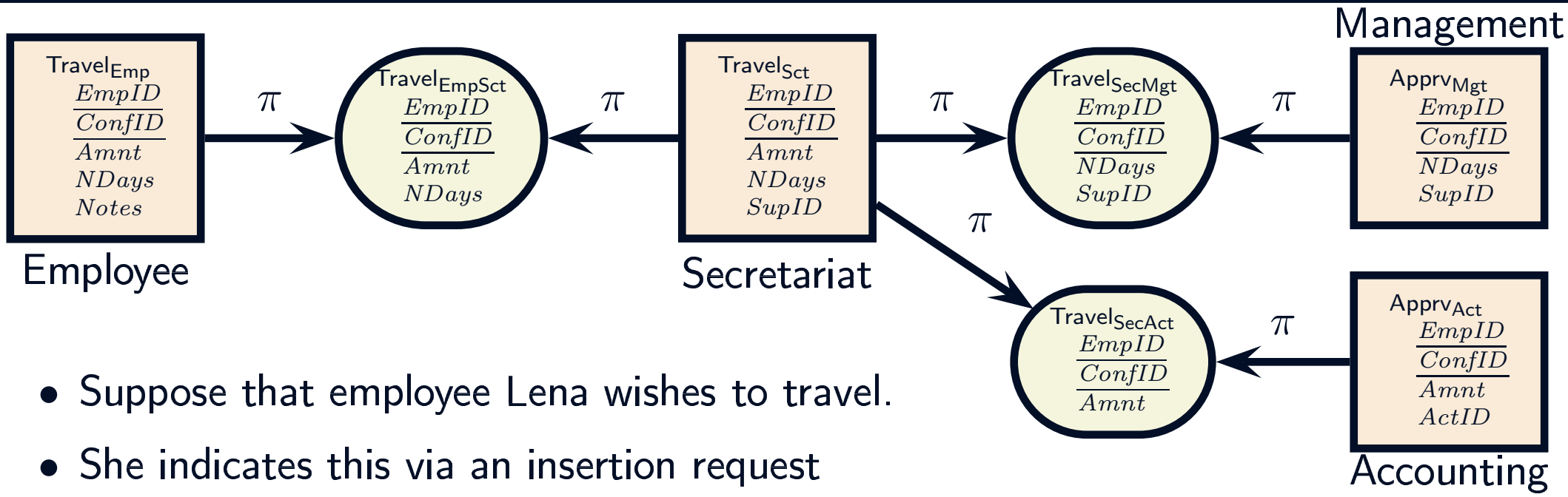

- Suppose that employee Lena wishes to travel.
- She indicates this via an insertion request into the schema of component Employee.

# 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.

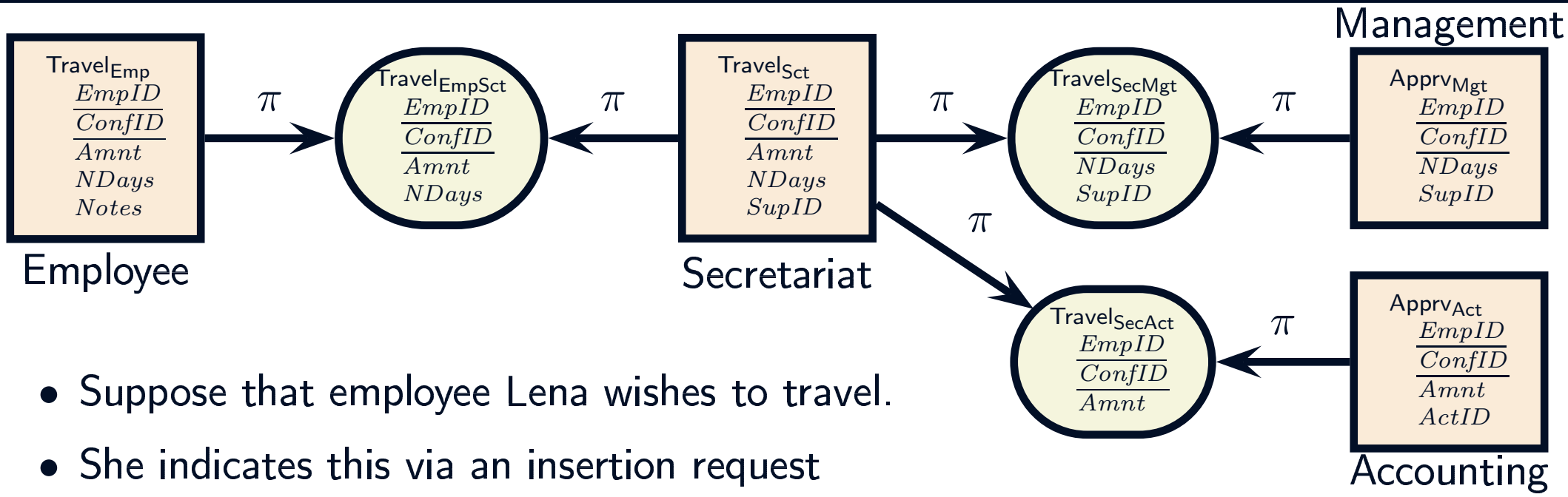
# Example: Travel Request and Authorization




- 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.

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

# Example: Travel Request and Authorization

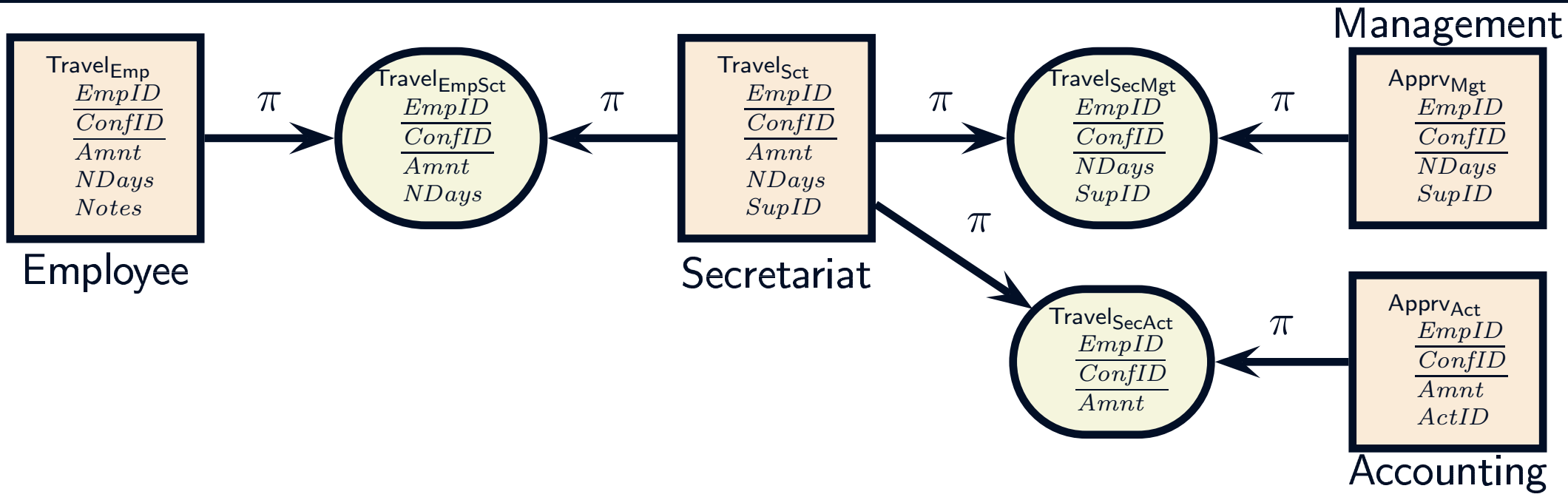



- Suppose that employee Lena wishes to travel.
- She indicates this via an insertion request into the schema of component Employee.
- Such a request is typically in the form of a finite *ranked set* of alternatives.

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

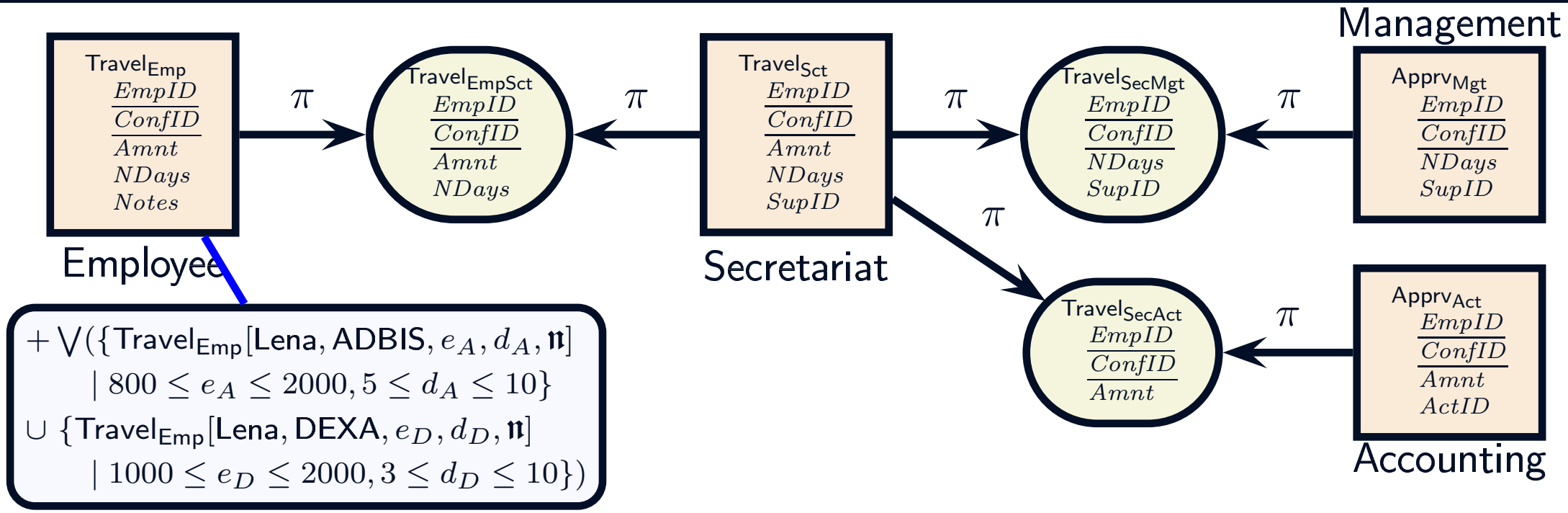
  - ADBIS is always preferred to DEXA.
  - More money and days are always preferred to fewer.

# Example: Evolution of Travel Request and Authorization



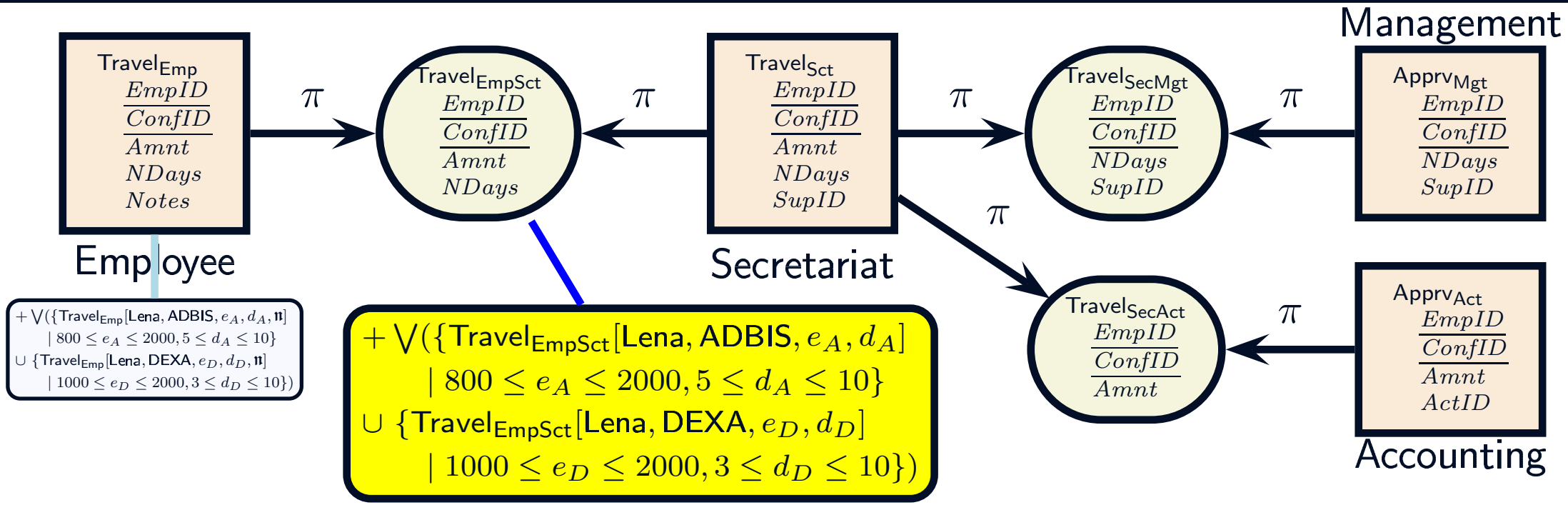

- The evolution of a specific update request will now be illustrated.

# Example: Evolution of Travel Request and Authorization



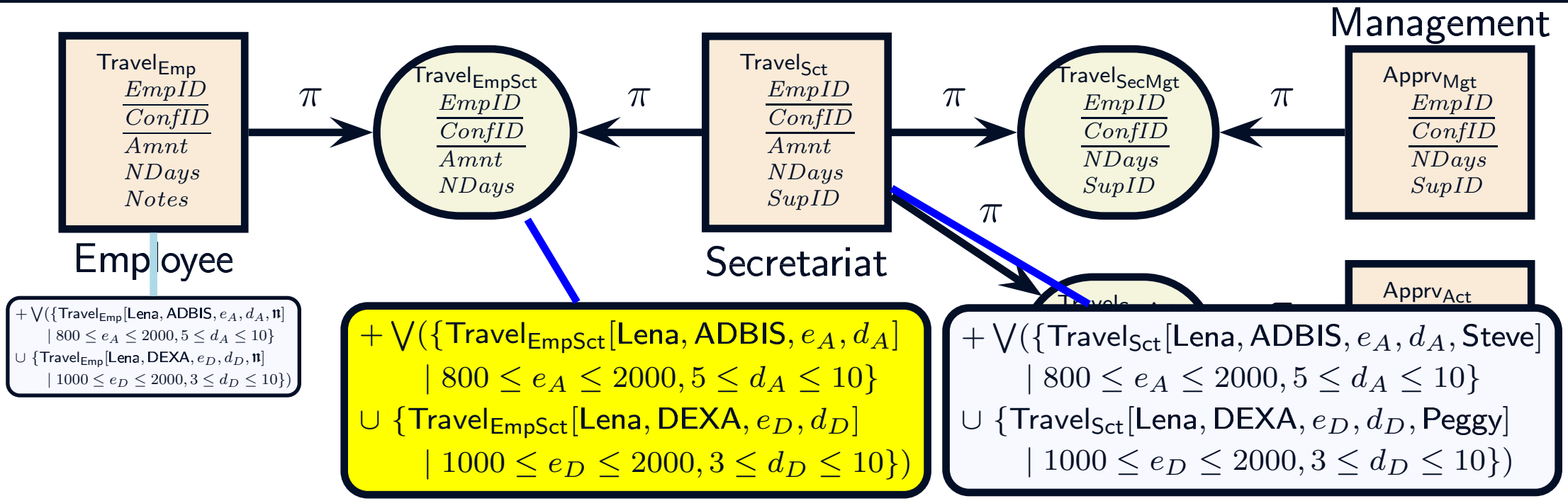

- First, the desired ranked update is entered into the pending update register for Employee. Notation:
  - $+$ = Insert.
  - $\bigvee$ = Choose one of the alternatives.

# Example: Evolution of Travel Request and Authorization



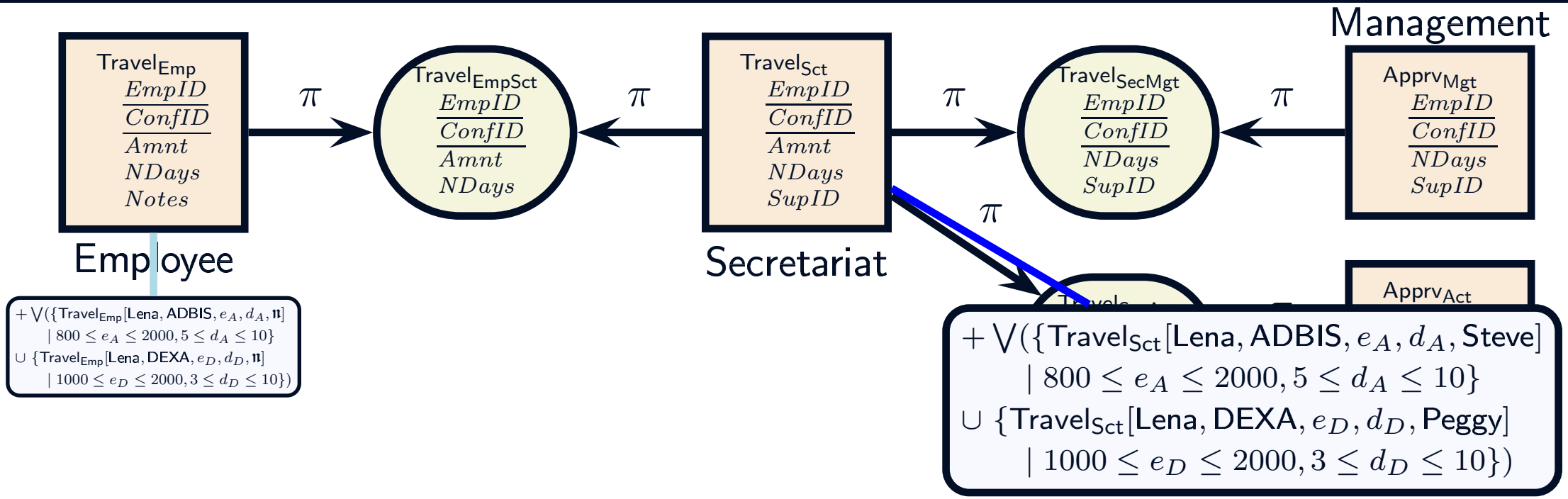

- This update is then projected to the port-status register which connects Employee to Secretariat.

# Example: Evolution of Travel Request and Authorization



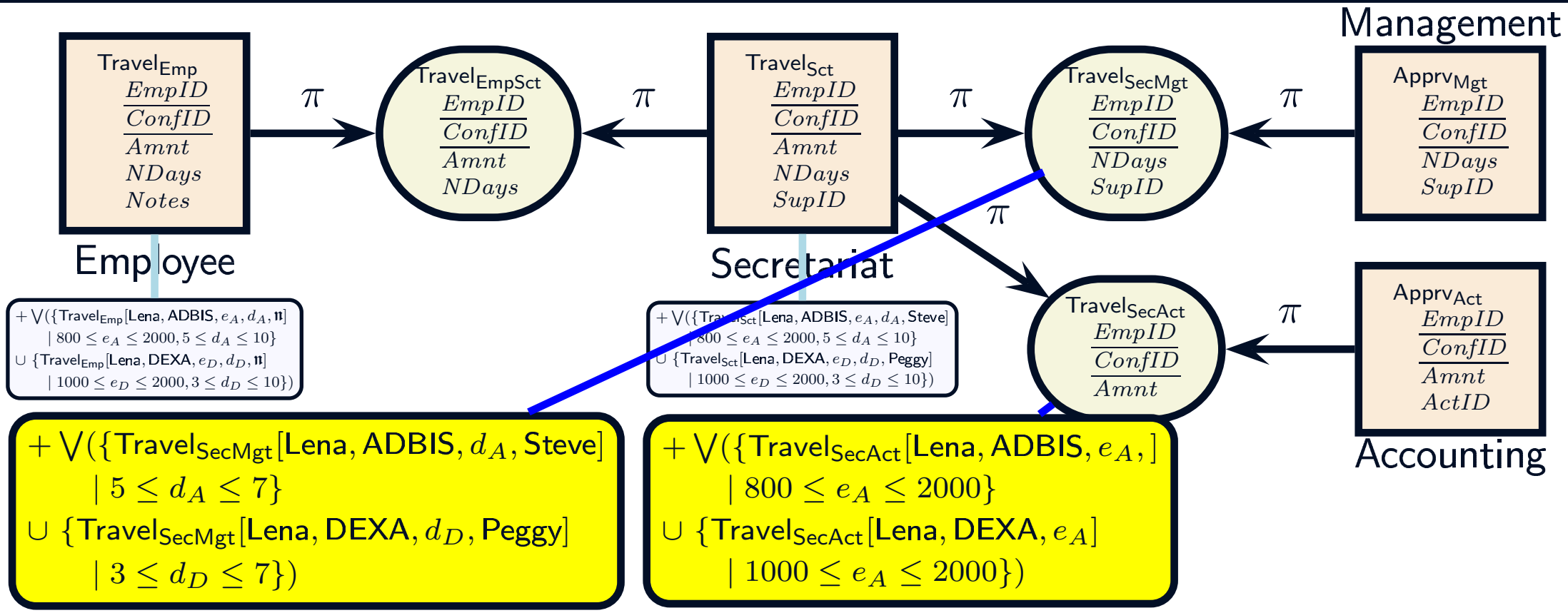

- The user of the Secretariat component then lifts this update to one on that component. It is placed in the pending-update register for that component.
  - Note that decisions must be made.
  - One of many possible liftings must be selected.

# Example: Evolution of Travel Request and Authorization



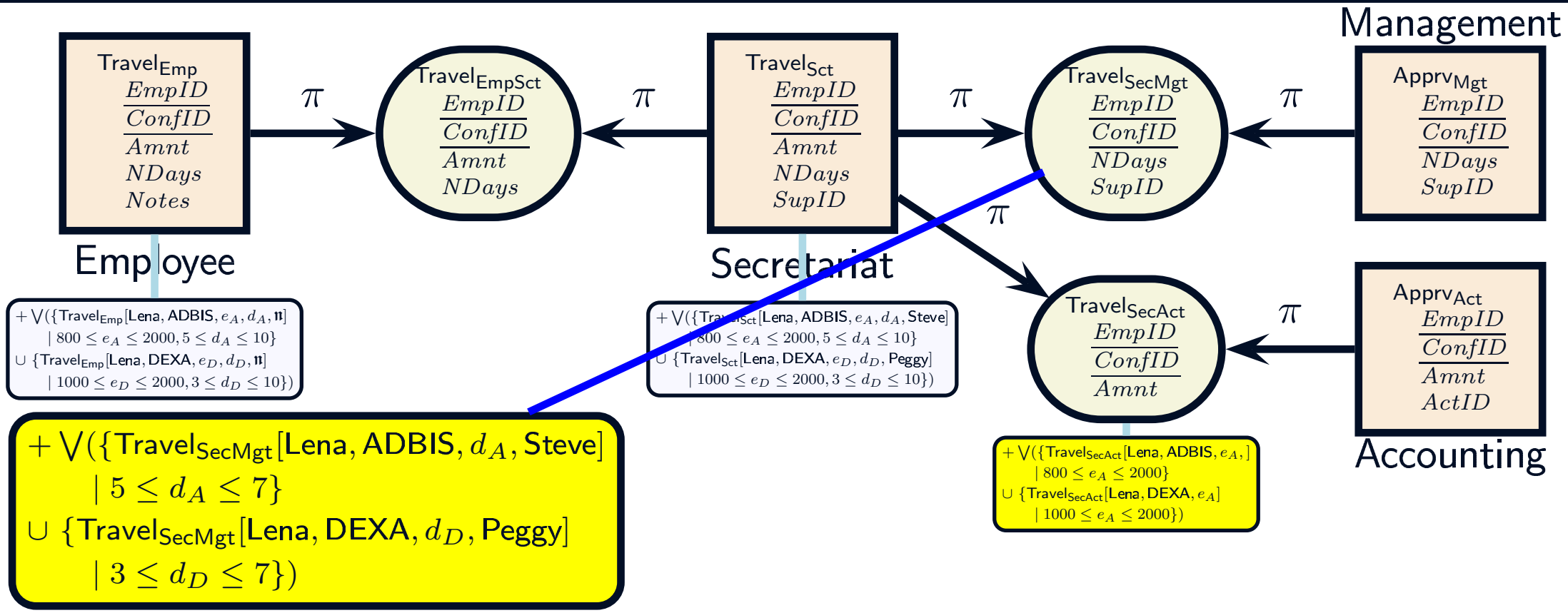

- The port-status register is then cleared, since this update has been processed.

# Example: Evolution of Travel Request and Authorization



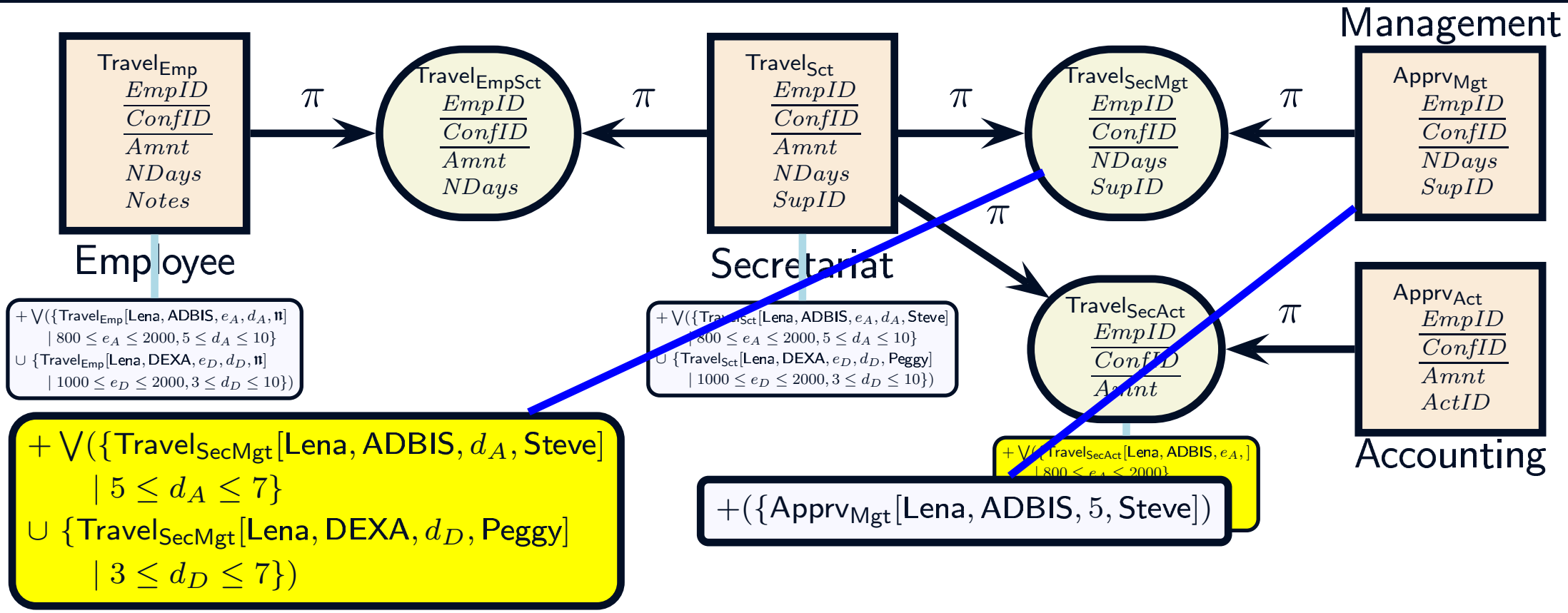

- This lifted update is then projected into the appropriate port-status registers which connect Secretariat to Management and Accounting.
- It is not projected back onto the port-status register which is connected to Employee, because the new value would be the same as the old one.

# Example: Evolution of Travel Request and Authorization



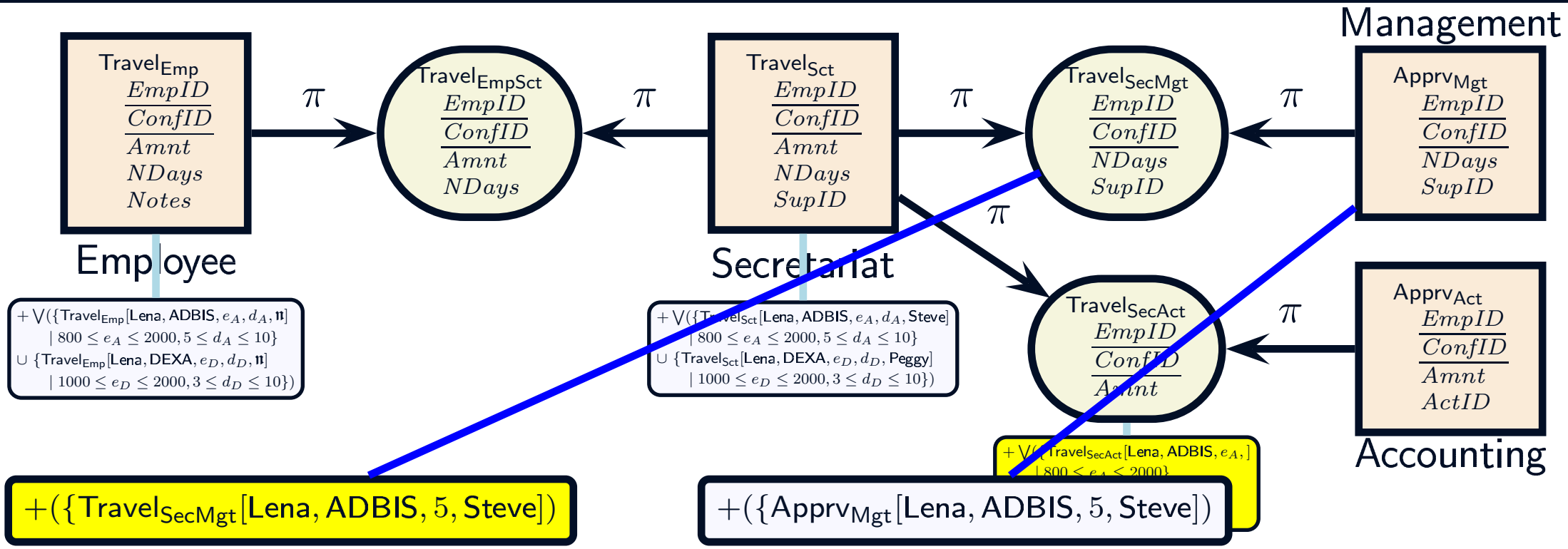

- First consider the problem of lifting the projected update to the Management component.
  - Again, there are decisions to be made.

# Example: Evolution of Travel Request and Authorization



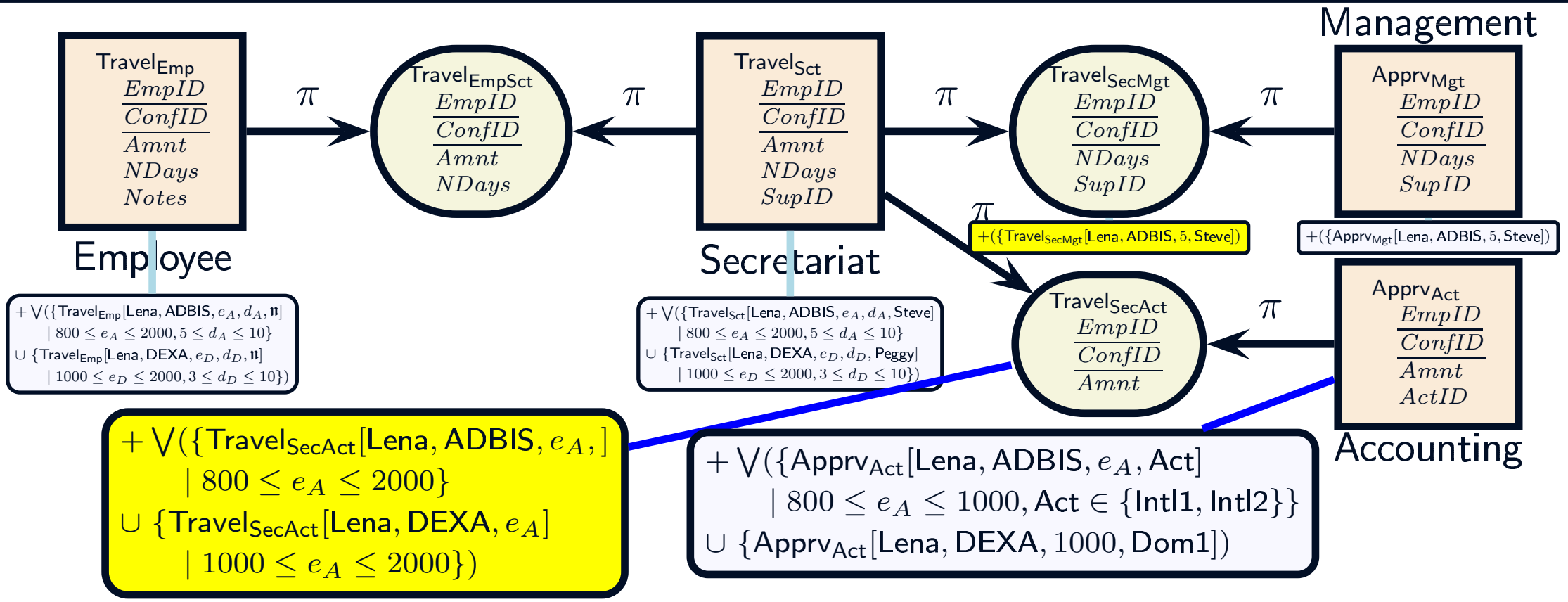

- Manager Steve processes the request, and decides to allow Lena to attend ADBIS for five days.

# Example: Evolution of Travel Request and Authorization



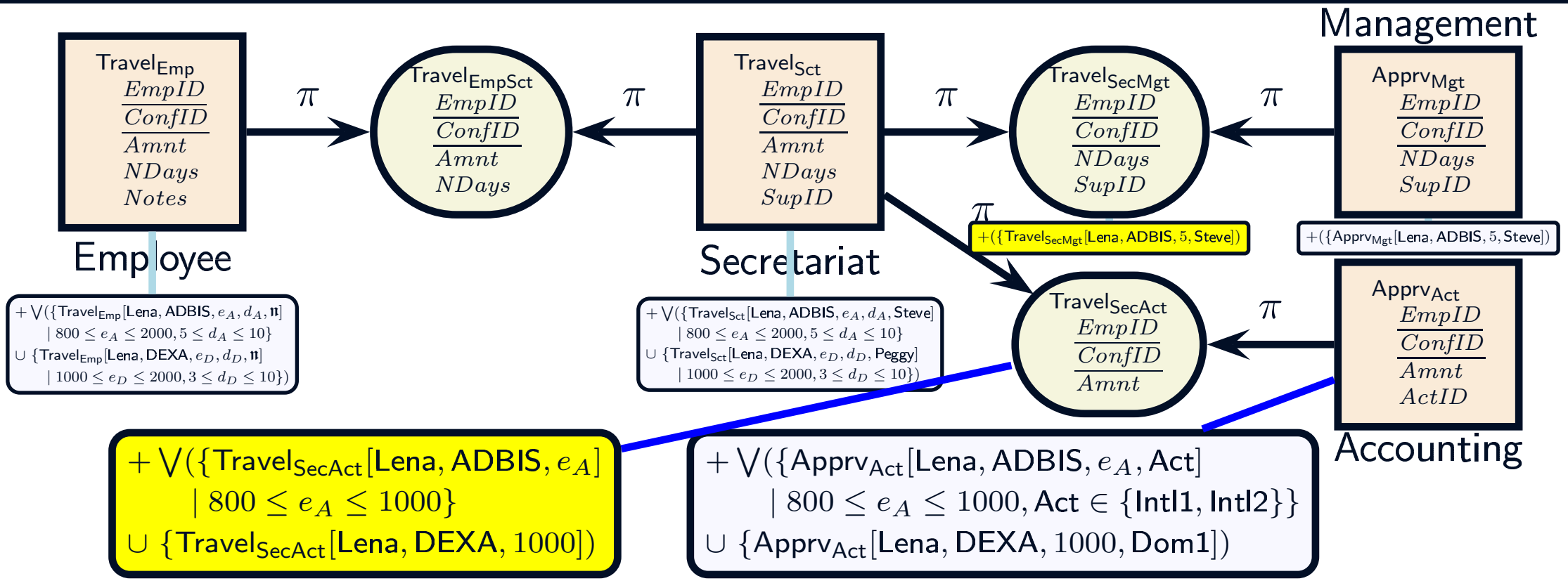

- The value in the port-status register for the Management component is removed, but a new value for the port-status register for Secretariat is inserted.

# Example: Evolution of Travel Request and Authorization



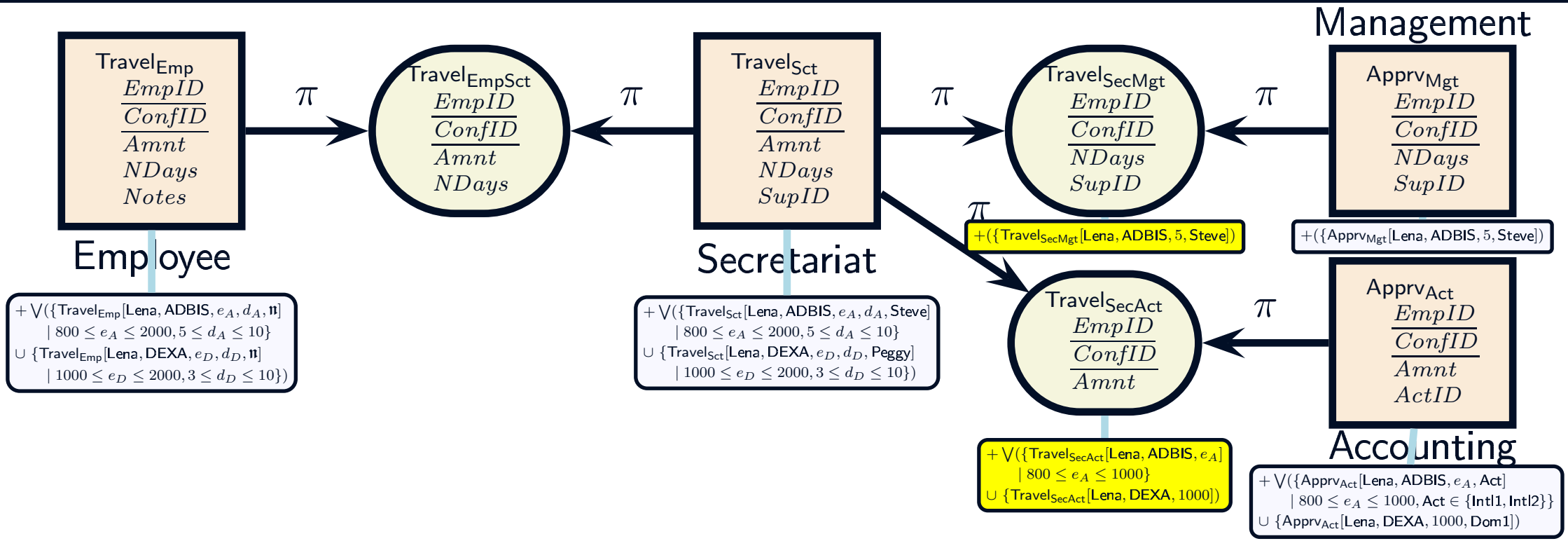

- Now consider the problem of lifting the projected update to the Management component.
  - The accounting manager decides to award €1000 for the requested travel.
  - The appropriate accounts are also identified.

# Example: Evolution of Travel Request and Authorization



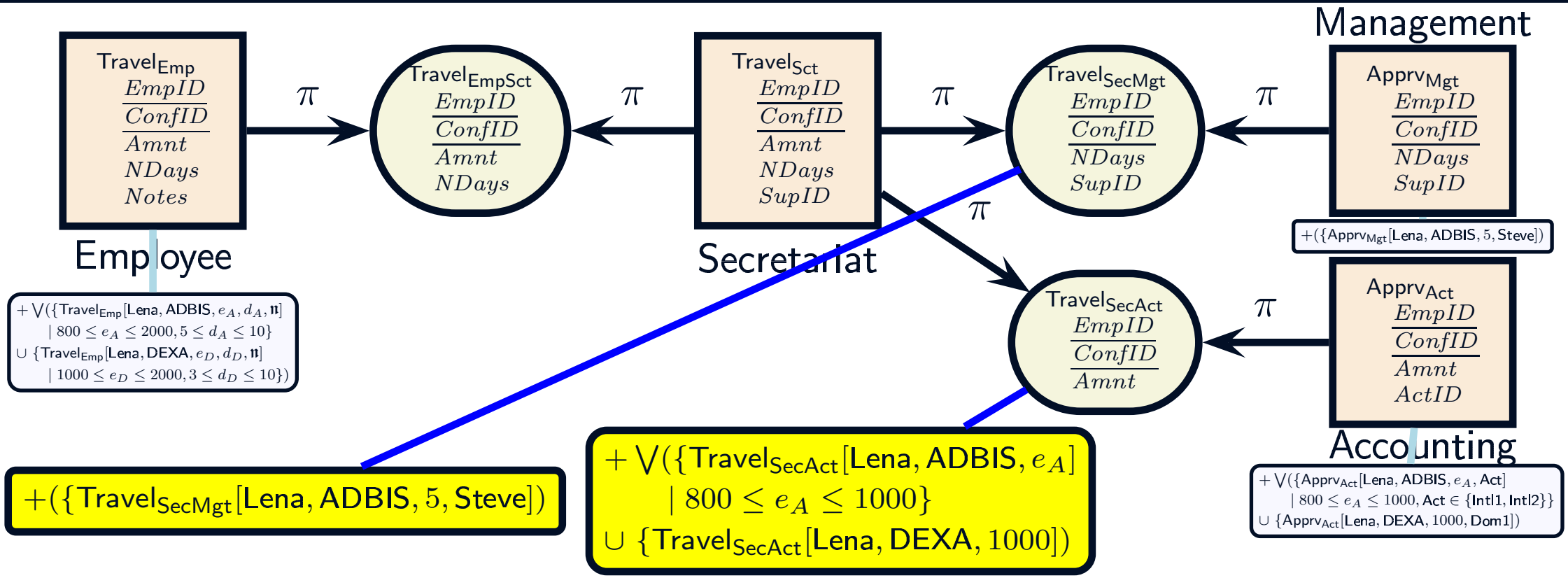

- The value in the port-status register for the Accounting component is cleared, but a new value for the port-status register for Secretariat is inserted.

# Example: Evolution of Travel Request and Authorization



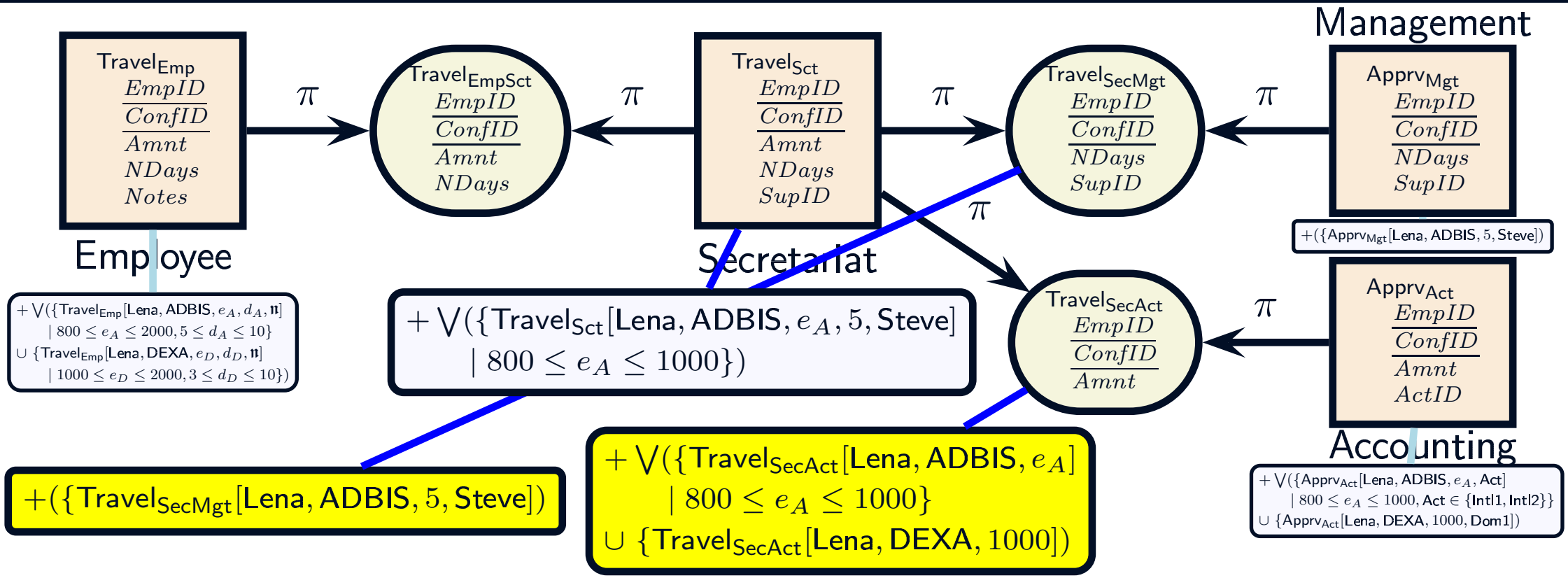

- Now the update negotiation propagates back right to left.

# Example: Evolution of Travel Request and Authorization



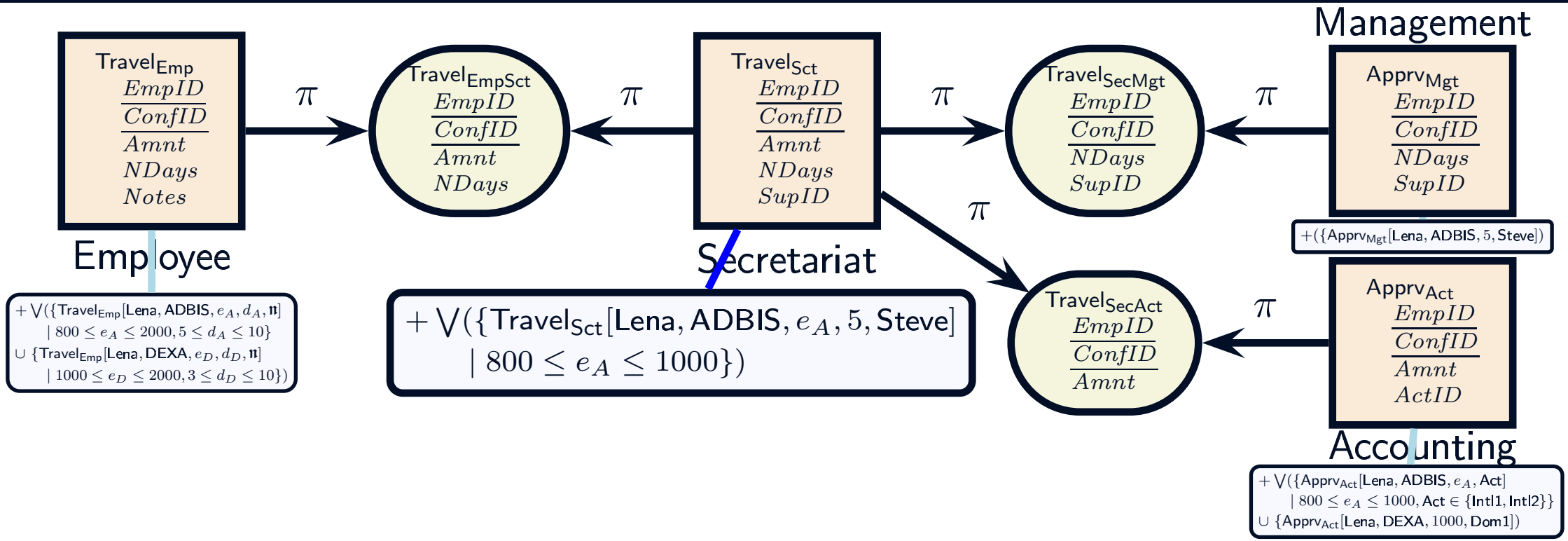

- The two values in the port-status registers must be lifted to the Secretariat component simultaneously.

# Example: Evolution of Travel Request and Authorization



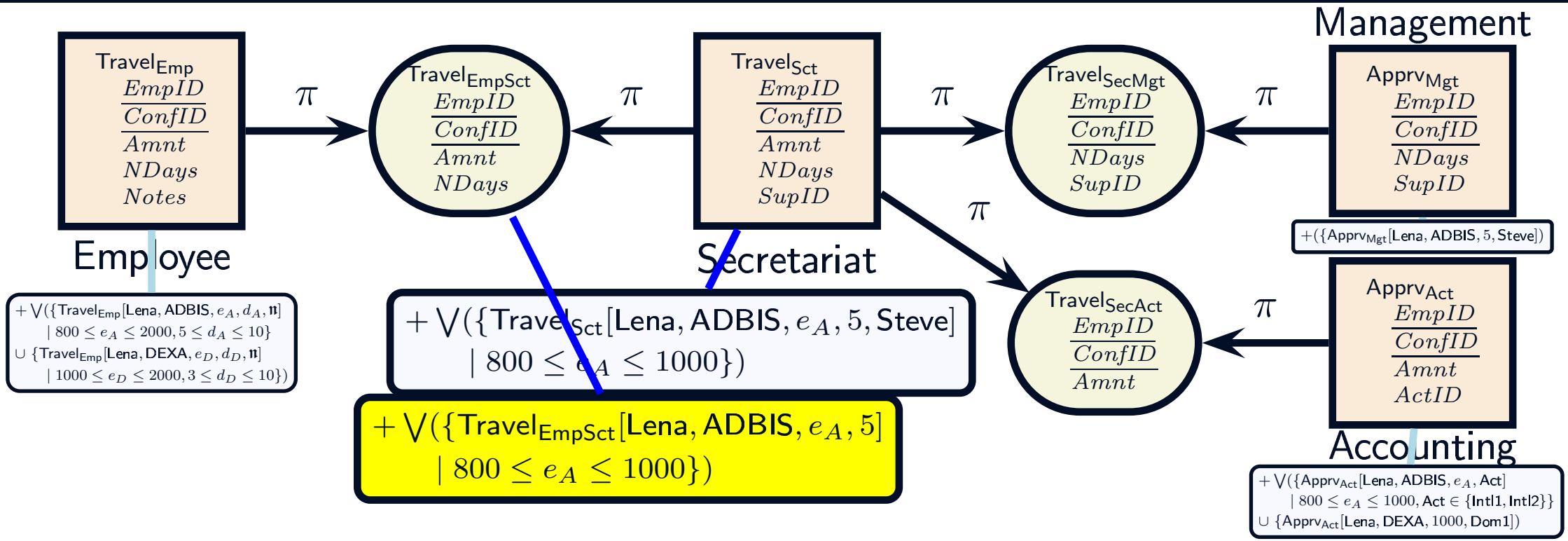

- Here is the maximal lifting.
  - The Secretariat makes imposes no additional limitations.

# Example: Evolution of Travel Request and Authorization



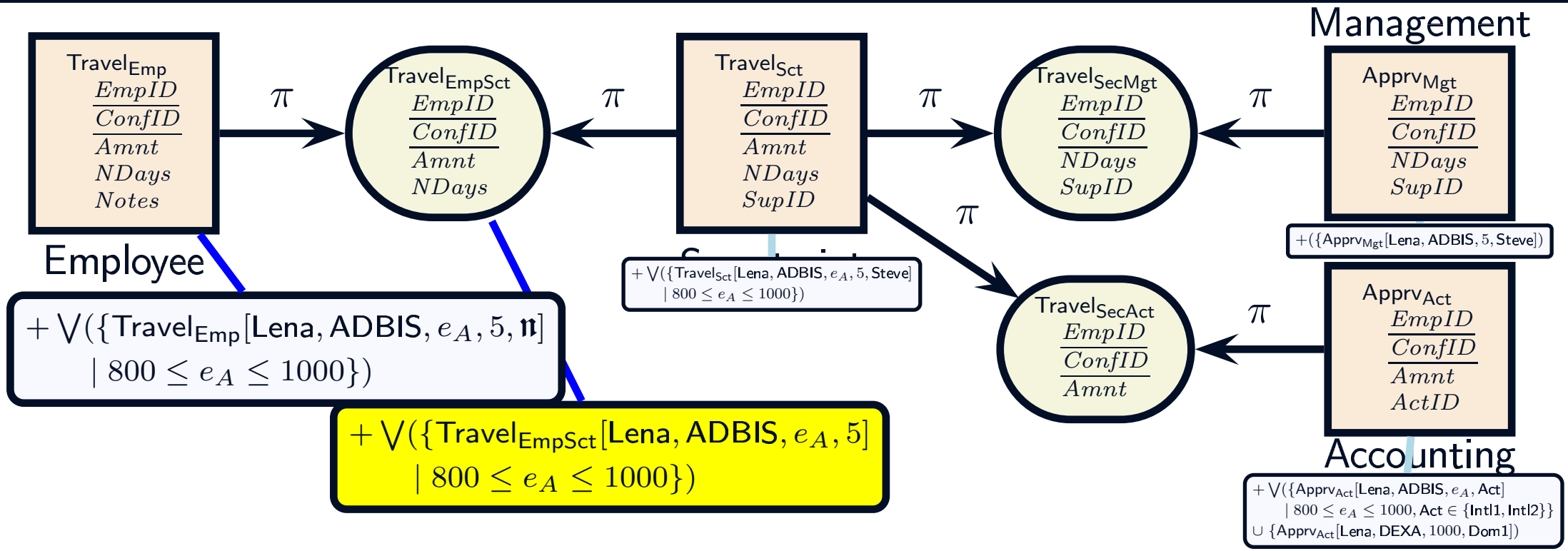

- The two port-status registers are now cleared.

# Example: Evolution of Travel Request and Authorization



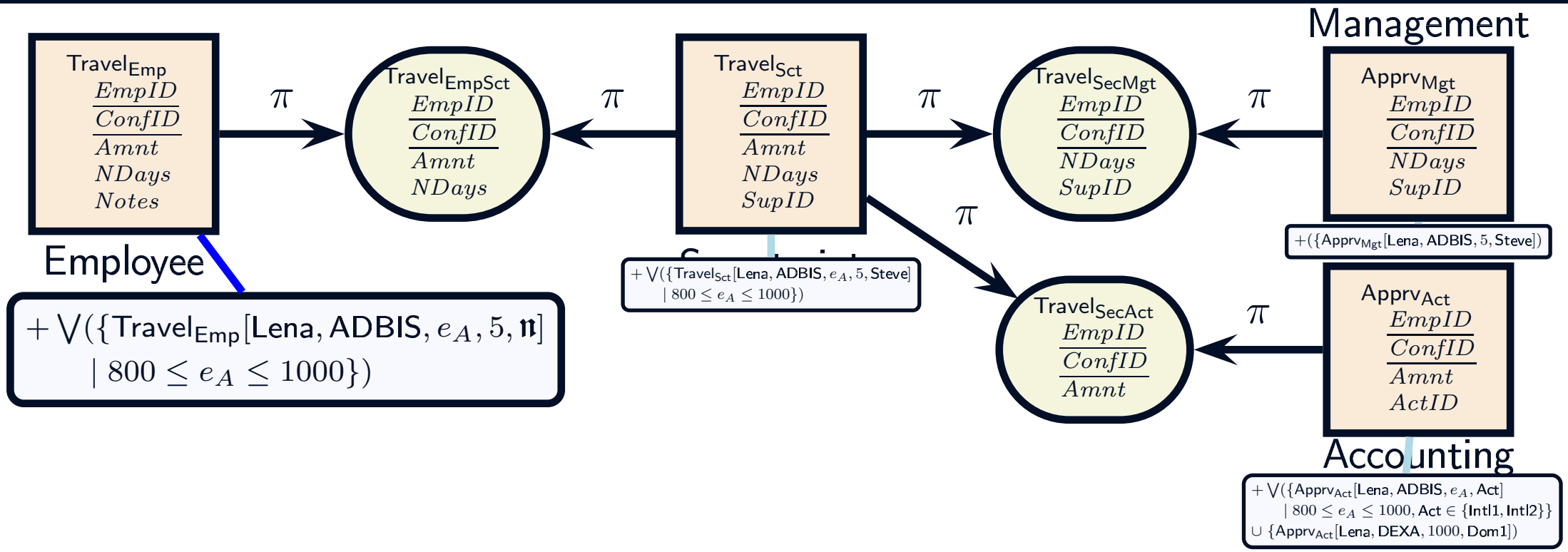

- The update request is then projected to the appropriate port-status register connecting Secretariat to Employee.

# Example: Evolution of Travel Request and Authorization



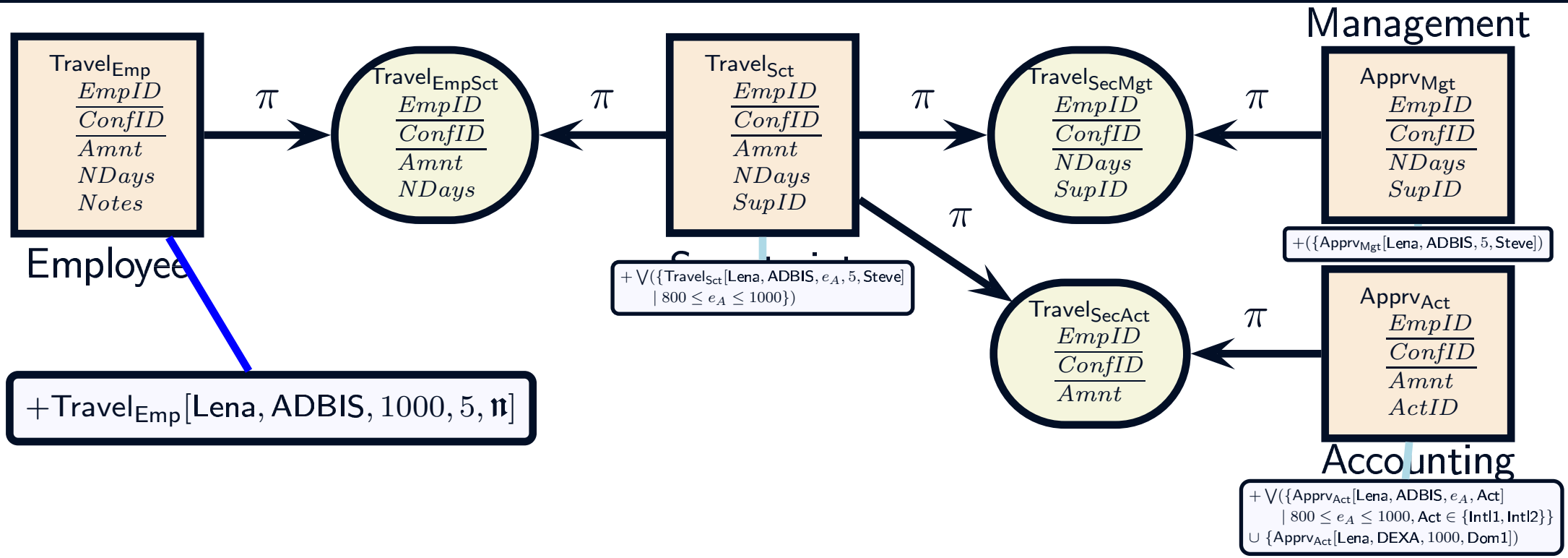

This update request is then lifted to the Employee component.

# Example: Evolution of Travel Request and Authorization



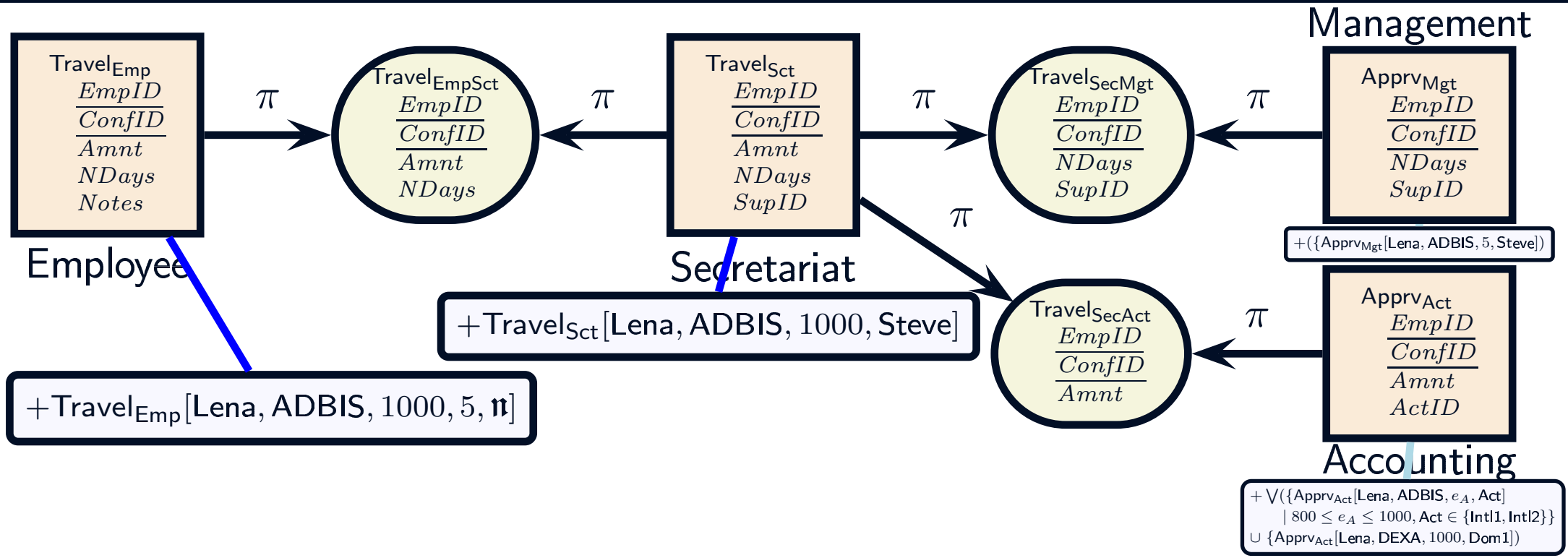

- The port-status register is cleared.
- Note that all port-status registers are now clear.

# Example: Evolution of Travel Request and Authorization



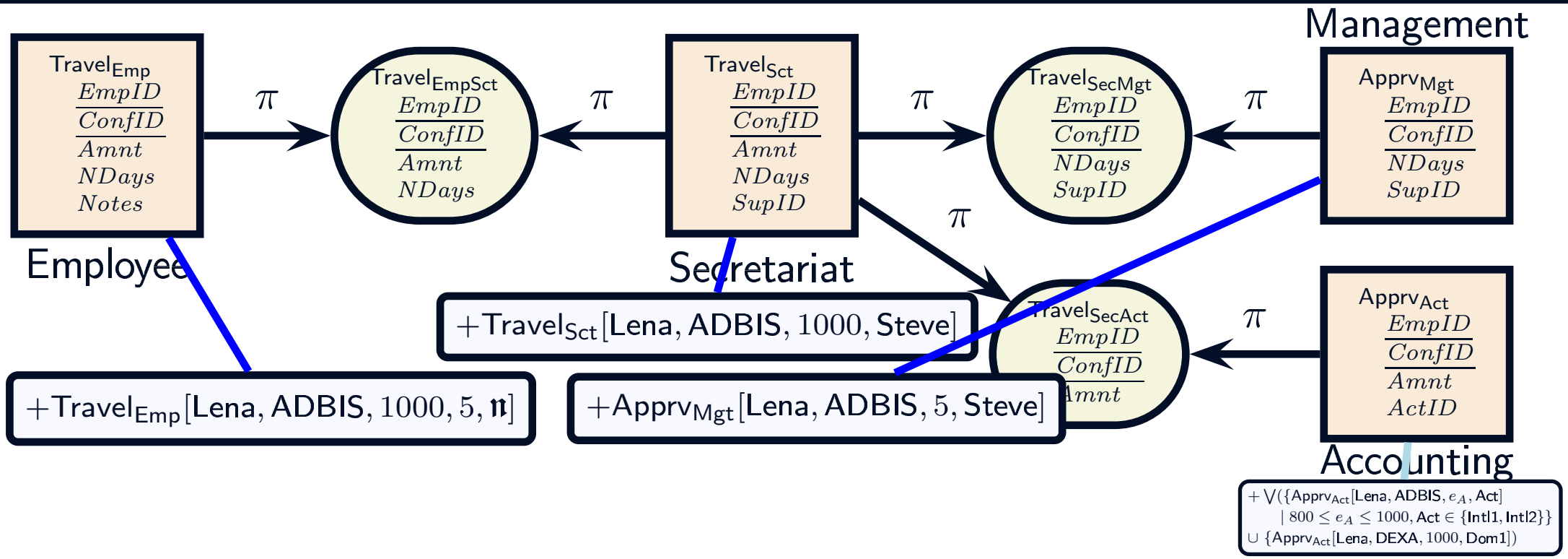

- Finally, Lena selects an update from amongst the possibilities.

# Example: Evolution of Travel Request and Authorization



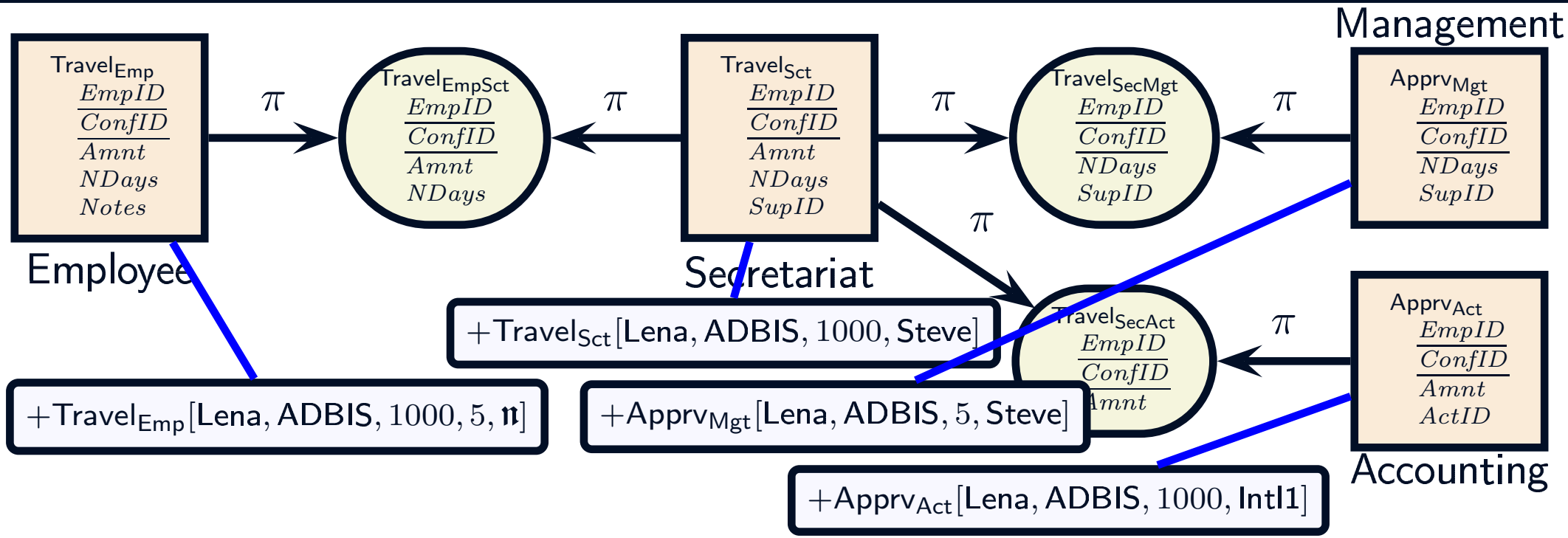

- This update is propagated to the other components for agreement.
- The messages passed through the port-status registers are not shown.

# Example: Evolution of Travel Request and Authorization



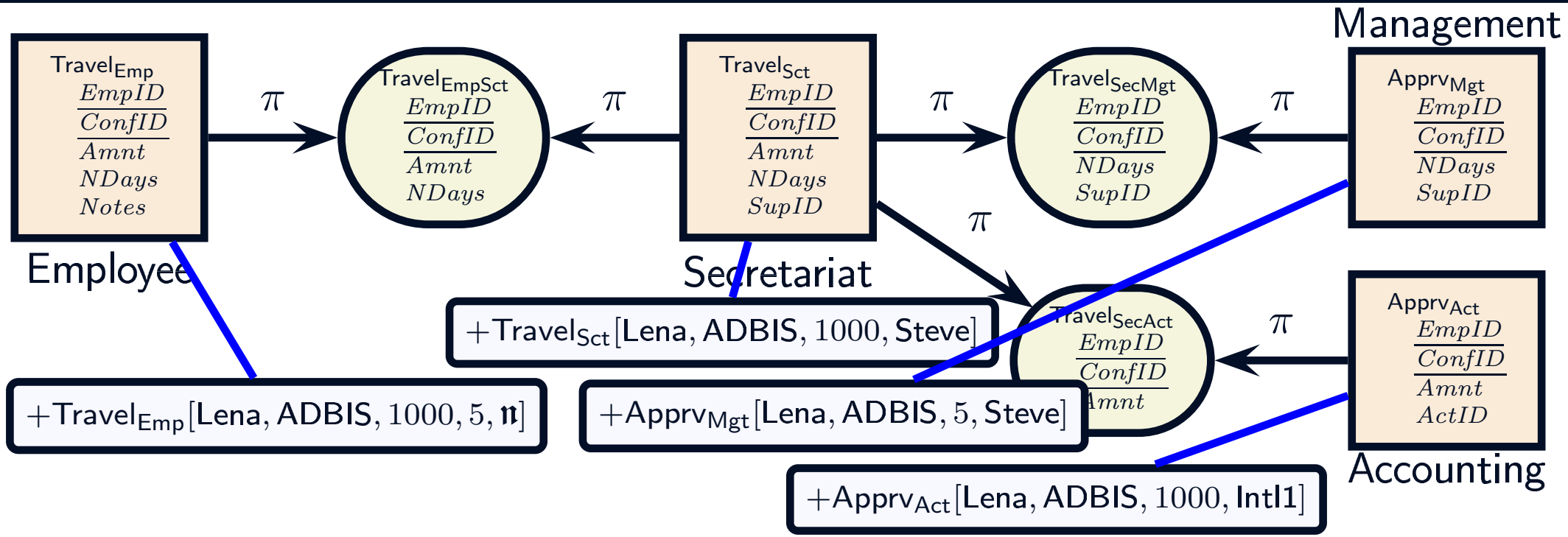

- This update is propagated to the other components for agreement.
- The messages passed through the port-status registers are not shown.

# 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.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.
- The *formal* model is a first, proof-of-concept design, and has the following limitations:

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.
- The *formal* model is a first, proof-of-concept design, and has the following limitations:
  - It is *opportunistic*: Individual users cannot control the flow of cooperation.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.
- The *formal* model is a first, proof-of-concept design, and has the following limitations:
  - It is *opportunistic*: Individual users cannot control the flow of cooperation.
  - It applies only to insertions and deletions.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.
- The *formal* model is a first, proof-of-concept design, and has the following limitations:
  - It is *opportunistic*: Individual users cannot control the flow of cooperation.
  - It applies only to insertions and deletions.
  - Negotiation is *monotonic*, in that update proposals can only be refined; additional changes cannot be added after the process begins.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.
- The *formal* model is a first, proof-of-concept design, and has the following limitations:
  - It is *opportunistic*: Individual users cannot control the flow of cooperation.
  - It applies only to insertions and deletions.
  - Negotiation is *monotonic*, in that update proposals can only be refined; additional changes cannot be added after the process begins.
- These limitations have the following positive implication.

# Conclusions and Properties of the Solution Technique

- A model for updating database views which is based upon the cooperation of interconnected components has been presented.
- This approach is particularly attractive in situations in which the reflected update requires access privileges beyond those possessed by the user of the view to be updated.
- The *formal* model is a first, proof-of-concept design, and has the following limitations:
  - It is *opportunistic*: Individual users cannot control the flow of cooperation.
  - It applies only to insertions and deletions.
  - Negotiation is *monotonic*, in that update proposals can only be refined; additional changes cannot be added after the process begins.
- These limitations have the following positive implication.

**Theorem** The negotiation process always terminates. Negotiations which proceed indefinitely are not possible. □

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

**Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

- **Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.
- **User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

- **Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.
- **User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

**Locking and implementation mechanism**:

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

**Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.

**User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

**Locking and implementation mechanism**:

- Multi-user cooperative update requires a suitable locking mechanism.

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

**Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.

**User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

**Locking and implementation mechanism**:

- Multi-user cooperative update requires a suitable locking mechanism.
- Implementation of component-based schemata also requires further study.

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

**Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.

**User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

**Locking and implementation mechanism**:

- Multi-user cooperative update requires a suitable locking mechanism.
- Implementation of component-based schemata also requires further study.

**Relationship to workflow**:

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

**Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.

**User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

**Locking and implementation mechanism**:

- Multi-user cooperative update requires a suitable locking mechanism.
- Implementation of component-based schemata also requires further study.

**Relationship to workflow**:

- There is an obvious close connection between the flow of control which cooperative update mandates and the notion of *workflow* for complex processes.

# Further Directions

**Complex negotiation**: The following extensions are particularly crucial.

**Nonmonotonicity**: A way to retract existing proposals and replace them with new ones.

**User-defined control flow**: A way to enable users to determine the flow of control in the negotiation process (in contrast to a purely opportunistic model).

**Locking and implementation mechanism**:

- Multi-user cooperative update requires a suitable locking mechanism.
- Implementation of component-based schemata also requires further study.

**Relationship to workflow**:

- There is an obvious close connection between the flow of control which cooperative update mandates and the notion of *workflow* for complex processes.
- The precise way in which cooperative update defines constraints on the possible workflow patterns for the system warrants further study.