Team automata : a formal approach to the modeling of collaboration between system components

Beek, M.H. ter

Citation

Beek, M. H. ter. (2003, December 10). *Team automata : a formal approach to the modeling of collaboration between system components*. Retrieved from https://hdl.handle.net/1887/29570

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/29570

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/29570 holds various files of this Leiden University dissertation.

Author: Beek, Maurice H. ter
Title: Team automata : a formal approach to the modeling of collaboration between system components
Issue Date: 2003-12-10

8. Applying Team Automata

In this chapter we give an impression of how team automata may be applied. We do this by presenting — in a varying degree of detail — three examples, each of which shows the usefulness of team automata in the early phases of system design. Additionally, we would like to mention that in [BLP03] we have initiated the use of team automata for the security analysis of multicast and broadcast communication. To this aim, team automata were used to model an instance of a particular stream signature protocol, while a well-established theory for defining and verifying a variety of security properties was reformulated in terms of team automata.

First we show — at a high level of abstraction — how to model a specific groupware architecture by team automata. To this aim we explain how team automata can be used as building blocks by internalizing certain external actions in order to prohibit their further use on a higher level of the construction (without changing the behavior of course).

Secondly, we show how team automata can be employed to model collaboration between teams of developers engaged in the development of models of complex (software) systems. This thus provides an example of using team automata for modeling interaction between humans. However, we still abstract from any social aspects and informal unstructured activity between humans. The team automata model solely the collaboration between humans.

Thirdly, we present a more detailed example demonstrating the potential of team automata for capturing information security and protection structures, and critical coordinations between these structures. On the basis of a spatial access metaphor, various known access control strategies are formally specified in terms of synchronizations in team automata. In [BB03] we have initiated an attempt to validate some of the resulting specifications with the model checker SPIN (see, e.g., [Hol91], [Hol97], and [Hol03]).

8.1 Groupware Architectures

In this section we show how team automata can be employed to model groupware architectures. To this aim we first introduce some notions and operations that are particularly useful when team automata are used for component-based system design. Consequently we use these operations to model a specific groupware architecture.

Notation 23. *Within this section we once again assume a fixed, but arbitrary and possibly infinite index set* $\mathcal{I} \subseteq \mathbb{N}$*, which we will use to index the component automata involved. For each* $i \in \mathcal{I}$*, we let* $\mathcal{C}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$ *be a fixed component automaton and we use* $\Sigma_{i,ext}$ *to denote its set of external actions* $\Sigma_{i,inp} \cup \Sigma_{i,out}$*. Moreover, we once again let* $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ *be a fixed composable system and we let* $\mathcal{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ *be a fixed team automaton over* $\mathcal{S}$*. Furthermore, we use* Σ *to denote the set of actions* $\Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ *and we use* Σ_{ext} *to denote the set of external actions* $\Sigma_{inp} \cup \Sigma_{out}$ *of any team automaton over* $\mathcal{S}$*. Recall that* $\mathcal{I} \subseteq \mathbb{N}$ *implies that* $\mathcal{I}$ *is ordered by the usual* $\leq$ *relation on* $\mathbb{N}$*, thus inducing an ordering on* $\mathcal{S}$*, and that the* $\mathcal{C}_i$ *are not necessarily different.* □

8.1.1 Team Automata as Architectural Building Blocks

As we have seen, a team automaton over a composable system is itself a component automaton that can be used in further constructions of team automata. Team automata can thus be used as building blocks. Before a team automaton is used as a building block, however, it may be necessary to internalize certain external actions in order to prohibit their further use on a higher level of the construction. The operation of hiding makes certain external actions of a component automaton invisible to other component automata by turning these external actions into internal actions. This operation has also been defined for I/O automata (see, e.g., [Tut87]).

Definition 8.1.1. *Let* $\mathcal{C} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ *be a component automaton and let* Δ *be an alphabet disjoint from* P*. Then*

> *the* Δ-hiding version *of* $\mathcal{C}$ *is denoted by* $\mathcal{C}_H^\Delta$ *and is defined as* $\mathcal{C}_H^\Delta = (P, (\Gamma_{inp} \setminus \Delta, \Gamma_{out} \setminus \Delta, \Gamma_{int} \cup \Delta), \gamma, J)$*.* □

Composability is in general not preserved by the operation of hiding since composability requires the internal actions of the component automata to belong to one component automaton only, whereas external actions are not subject to such a restriction. The Δ-hiding version of a team automaton

thus need not be a team automaton over the Δ-hiding versions of its original constituting component automata. For our composable system $\mathcal{S}$ and subsets $\Delta_i \subseteq \Sigma_{i,ext}$, for all $i \in \mathcal{I}$, the system $\mathcal{S}' = \{(\mathcal{C}_i)_H^{\Delta_i} \mid i \in \mathcal{I}\}$ is composable if and only if for all $i \in \mathcal{I}$, $\Delta_i \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_{j,ext} = \varnothing$.

The external actions that are to be hidden are those that are only used for communications between certain component automata and that should not be available for communication with other component automata.

Definition 8.1.2. *A pair $\mathcal{C}_i, \mathcal{C}_j$, with $i, j \in \mathcal{I}$, is communicating (in $\mathcal{S}$) if there exists an $a \in (\Sigma_{i,ext} \cup \Sigma_{j,ext})$ such that*

$$a \in (\Sigma_{i,inp} \cap \Sigma_{j,out}) \cup (\Sigma_{j,inp} \cap \Sigma_{i,out}).$$

Such an a is called a communicating action (in $\mathcal{S}$). By Σ_{com} we denote the set of all communicating actions (in $\mathcal{S}$). □

Note that the *communicating relation* between component automata, i.e. the set of all pairs of communicating component automata over component automata, is symmetric and irreflexive. Note furthermore that the fact that an action is communicating does not imply that a team automaton over $\mathcal{S}$ will actually have a synchronization involving this action as a communication, i.e. in its two roles of input and output. The communicating property is based solely on alphabets and is thus by no means related to transition relations.

With the hide operation we can internalize all communicating actions of a team automaton, before this team automaton is used to build a higher-level team automaton. The result is a team automaton that is closed with respect to its communications to the outside world.

Definition 8.1.3. *The* (communication) closed version *of $\mathcal{T}$ is denoted by $\boxed{\mathcal{T}}$ and is defined as*

$$\boxed{\mathcal{T}} = \mathcal{T}_H^{\Sigma_{com}}.$$ □

Rather than the team automaton itself we may now use its closed version in a new construction. If we do this, then only those output (input) actions that do not have a matching input (output) action within the team automaton are external actions of the closed version of the team automaton. The remaining external actions have been reclassified as internal actions.

In practice one often wants to work with several copies of a component automaton. In our model, however, more than one copy of a component automaton in a set of component automata in general means that this set does not satisfy composability. An operation renaming the actions of a component

automaton solves this problem. Modulo renaming, these copies all have the same computations (and thus exhibit the same behavior). The operation of renaming has also been defined for I/O automata (see, e.g., [Tut87]).

Recall that a function $f : A \to A'$ is a bijection if it is injective ($f(a_1) \neq f(a_2)$ whenever $a_1 \neq a_2$) and surjective (for every $a' \in A'$ there exists an $a \in A$ such that $f(a) = a'$).

Definition 8.1.4. *Let $\mathcal{C} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be a component automaton, let Δ be an alphabet disjoint from P, and let $h : (\Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}) \to \Delta$ be a bijection. Then*

> *the* h-renamed version *of $\mathcal{C}$ is denoted by $\mathcal{C}_N^h$ and is defined as $\mathcal{C}_N^h = (P, (h(\Gamma_{inp}), h(\Gamma_{out}), h(\Gamma_{int})), \{(q, h(a), q') \mid (q, a, q') \in \gamma\}, J)$.* □

In practice, an h-renamed version of a component automaton might best be defined to generate new names which are disjoint from the domain set, e.g. by requiring Δ to be disjoint from its alphabet.

It is clear that, apart from the use of new names, certain properties of team automata continue to hold for their h-renamed versions.

Lemma 8.1.5. *Let h be a bijection such that $\mathcal{T}_N^h$ is the h-renamed version of $\mathcal{T}$. Then*

(1) $\mathbf{C}^{\infty}_{\mathcal{T}_N^h} = \widehat{h}(\mathbf{C}^{\infty}_{\mathcal{T}})$, where $\widehat{h}$ is the extension of h to $\Sigma \cup Q$ defined by $\widehat{h}(q) = q$, for all $q \in Q$,

(2) $\mathbf{B}^{\Sigma,\infty}_{\mathcal{T}_N^h} = h(\mathbf{B}^{\Sigma,\infty}_{\mathcal{T}})$, and

(3) if an action a is free (ai, si, sipp, wipp, sopp, wopp, ms, sms, wms) in $\mathcal{T}$, then $h(a)$ is free (ai, si, sipp, wipp, sopp, wopp, ms, sms, wms) in $\mathcal{T}_N^h$. □

In the next subsection we show how to apply the operations introduced here.

8.1.2 GROVE Document Editor Architecture

In [Ell97] the distributed architecture of the GROVE document editor (see, e.g., [EGR90]) — depicted here in Figure 8.1 — is discussed. In this section we show how to model this architecture using a formal description in terms of team automata. In the process we point out where the notions introduced in the previous subsection come into play.

We are given a user interface automaton $\mathcal{C}_1$, a keeper automaton $\mathcal{C}_2$, an application automaton $\mathcal{C}_3$, and a coordination automaton $\mathcal{C}_4$. These together form a composable system $\mathcal{S} = \{\mathcal{C}_i \mid i \in [4]\}$. Only the pairs $\mathcal{C}_i, \mathcal{C}_{i+1}$, $i \in [3]$,

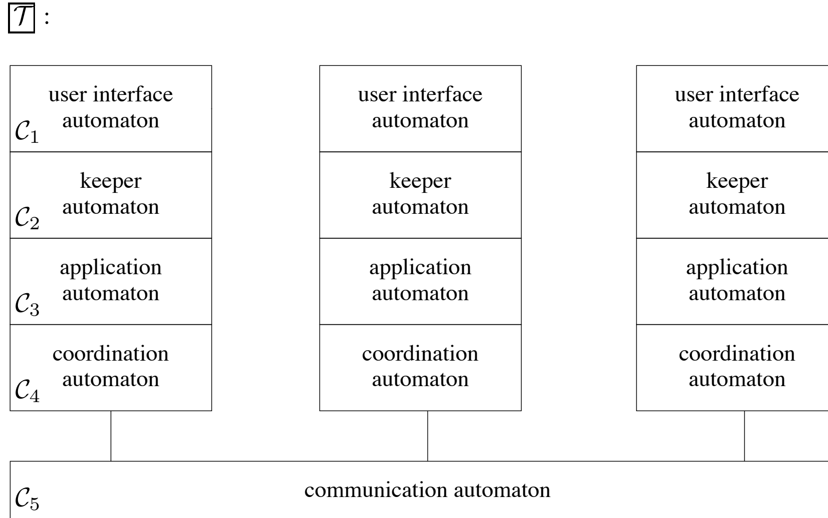

Fig. 8.1. The GROVE document editor architecture.

are communicating. All external actions of $\mathcal{C}_2$ and $\mathcal{C}_3$ are communicating in $\mathcal{S}$. $\mathcal{C}_1$ has external actions that are not communicating in $\mathcal{S}$, but intended to be used solely for interaction with the users. $\mathcal{C}_4$ has external actions to be used for communication with the communication automaton $\mathcal{C}_5$, which is to be added in a later stage. However, the non-communicating actions of $\mathcal{C}_1$ are different from those of $\mathcal{C}_4$.

The architecture requires all components in $\mathcal{S}$ to synchronize on all communications, thus we construct the *maximal-ai*-team automaton $\mathcal{T}$ over $\mathcal{S}$. Then this team automaton $\mathcal{T}$ is closed, resulting in its closed version $\boxed{\mathcal{T}}$. Now all communicating external actions are internal in $\boxed{\mathcal{T}}$. In this way we prohibit further synchronizations involving a component of $\mathcal{S}$. The only remaining external actions are those of $\mathcal{C}_1$ and those of $\mathcal{C}_4$.

Next we introduce several renamed versions of $\boxed{\mathcal{T}}$ satisfying the following two conditions.

First, the sets of actions of the renamed versions should be mutually disjoint in order to avoid undesired synchronizations of their user interfaces, and of actions to be used for the interaction with the communication automaton $\mathcal{C}_5$. Note that this condition ensures that these renamed versions form a composable system $\mathcal{S}'$.

Secondly, the external actions of $\boxed{\mathcal{T}}$ originating from the coordination automaton $\mathcal{C}_4$ should be renamed in such a way that they will communicate with actions from $\mathcal{C}_5$.

Finally, to obtain the desired team automaton modeling the GROVE document editor architecture we define a team automaton over $\mathcal{S}'' = \{\mathcal{C}_5\} \cup \mathcal{S}'$. Since we want $\mathcal{C}_5$ to communicate with all renamed versions of $\boxed{\mathcal{T}}$ we construct the *maximal-ai*-team automaton over $\mathcal{S}''$, which thus results in all communicating actions being synchronized.

It is clear that the iterated way in which we have constructed this final team automaton guarantees that no undesired synchronizations between, e.g., a keeper automaton and the communication automaton can take place. Not only all communication between the communication automaton and any of the renamed versions of $\boxed{\mathcal{T}}$ takes place via their coordination automata, but also there are no interactions between the renamed versions of $\mathcal{T}$. This is conveniently modeled by the communication closure. Moreover, the explicit construction used to form the final team automaton makes all communications mandatory.

8.1.3 Conclusion

In this section we have seen how team automata can be used to model both the conceptual and the architectural level of groupware systems. Actually, many of the concepts and techniques of computer science, such as concurrency control, user interfaces, and distributed databases, need to be rethought in the groupware domain. Team automata are thus helpful for this rethinking. The team automata framework allows one to separately specify the components of a groupware system and to describe their interactions. It is thus neither a message-passing model nor a shared-memory model, but a shared-action model. In particular, we have seen that team automata provide us with tools allowing formal and precise definitions of various basic groupware notions.

One way of viewing the team automaton framework is as having a two-way mechanism to model a spectrum of group interactions. On the one hand we have peer-to-peer types of synchronization, in which all participants are considered equal. They model the group collaboration aspect that frequently occurs in synchronous groupware. On the other hand there are master-slave types of synchronization, in which output as a master may force the concurrent execution of a corresponding input action. They can be used to model asynchronous cooperation, as in workflow systems to enact certain modules (see, e.g., [EN93]).

Team automata thus fit nicely with the needs and the philosophy of groupware and thanks to the formal setup, theorems and methodologies from automata theory can be applied.

8.2 Team-Based Model Development

Software configuration management is a subfield of *software engineering* that deals with organizing and controlling evolving software systems throughout their life cycle (see, e.g., [IEEE93]). Through software configuration management models, technical and administrative direction and surveillance over the life cycle of software systems is given in order to identify the functional and physical characteristics of modules and their assemblies, to control releases and changes, to record the product status, and to validate the completeness, consistency, and conformance to specifications of the product. Incorporated are also areas such as construction management, process management, and team work control (see, e.g., [Dar91]).

Since software systems are becoming more and more complex, it is inevitable to parallelize the development of models for these systems in such a way that several teams of developers must work in parallel on (parts of) the model under design. At some point in time the efforts of these teams however need to be integrated and this, more often than not, leads to conflicts. Obviously, these conflicts need to be resolved. However, most of the time they are difficult and time consuming to resolve and furthermore they often require manual modeler intervention.

8.2.1 A Conflict-Free Cooperation Strategy

Software configuration management models use a *cooperation strategy* to ensure that changes are coordinated such that one change does not — unwillingly — undo or conflict with the effects of another change. A *conservative cooperation strategy* prevents conflicting changes by using a simple locking scheme: developers working on a specific module version or configuration can lock it against further changes, and while a version or configuration is locked other developers are excluded from creating new versions. On the contrary, in an *optimistic cooperation strategy* each developer is active in his or her own workspace and various versions of the same module can be created.

Both conservative and optimistic cooperation strategies eventually need to merge parallel changes. Existing approaches of merges often lack early conflict detection, which results in conflicts becoming apparent only during the actual merges. These conflicts then have to be resolved, which is very time consuming. A conservative cooperation strategy does reduce the potential number of conflicts, since each part of a model may only be changed by one team *at a time* (the situation where two or more teams are working at cross purposes is avoided). However, a change to one part can affect all dependent parts and unfortunately thus still lead to conflicts during merge.

We note that problems during merge are avoided if we have a precise definition of when a change to a part is local, i.e. when the change only affects that part and not the rest of the model. When using an optimistic strategy, each part is edited in its own workspace by one unique team of developers. If we thus require each team to make local changes only to its own part, then integration becomes straightforward and, in fact, can be done automatically due to the absence of conflicts. We call this a *conflict-free (cooperation) strategy*.

We now illustrate our conflict-free strategy for the development of an object-oriented model. As parts of the model we use packages of classes, which are commonly used to structure a model (see, e.g., [RBP$^+$91] and [UML99]). A notion of local change can, e.g., be defined through invariancy of the services offered through the interface of the package. The interface is then the contract of the package with the rest of the model ([Mey92]).

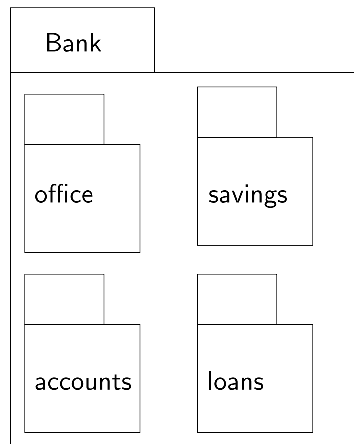

Fig. 8.2. The departments of a bank.

In Figure 8.2 we present part of a model in which a package Bank models a real-life bank (the figure is drawn using the notation of [UML99]). Four of its departments are modeled as subpackages. Bank can be developed in parallel by four teams where each team separately develops one of the departments. The changes made to each department are local and the merge to form the modified bank is straightforward. Note that these packages can be developed in entirely different geographic locations. Each team has its own workspace to make its changes and is only dependent on the other teams during merge.

We use an optimistic strategy, but we constrain the changes in each workspace to prevent conflicts during merge. A model is split into several views for individual development and later merge. In this case we however block changes to the views which cause conflict during merge. In any realistic project, however, the connections between the parts (packages) of the model cannot stay the same during the complete life cycle of the model. Modifications requiring non-local changes of packages (thus invalidating the conflict-free strategy) need to occur and hence a conflict-free merge cannot be guaranteed. These changes can however be localized by (temporarily) adding a new package, which contains those original packages between which changes have to be made. These changes are then local with respect to the newly added package and thus allow for the conflict-free strategy to be applied to the model with the extra package. This is illustrated in Figure 8.3.

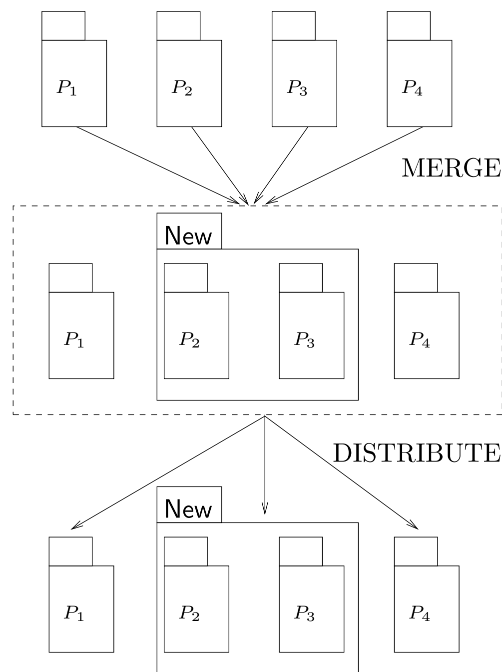

Fig. 8.3. A package is added.

The packages P_1 to P_4 are edited using the conflict-free strategy. However, non-local changes are required between packages P_2 and P_3. The work under development is merged and a temporary package New is added to group these two wayward parts. Note that because up to now the changes to the

packages of the model have been local, the merge is without conflicts. The model is consequently redistributed with the new structure and work can continue under the conflict-free strategy, since changes are once again local. The extra package can be removed once the new connections between the wayward packages are stable.

Note that in practice it may not be necessary to merge all the packages under development. It may be sufficient only to merge the packages for which the non-local changes are required to form a partial model, e.g., if each package is at most once the subject of such a temporary merge before a complete intermediary model is produced.

The architecture of a model thus is initially determined by top-down decomposition. This architecture can however be adapted to suit the need of our strategy. We call this part of the conflict-free strategy the *renegotiation phase*. Too many of such phases during the model's life cycle are inconvenient. They however indicate that the high-level architecture of the model is not yet stable, or even that the model is as yet too premature to be developed in a distributed fashion. Ideally, the initial breakdown of the model into packages should only be done by experienced modelers, thereby reducing the number of renegotiation phases as much as possible. The initial model should consequently be developed in one workspace until there is enough confidence that a right choice has been made for a stable enough architecture, after which the conflict-free strategy can be applied to it. The same considerations hold when one of the packages used in the conflic-free strategy is further split up into two or more subpackages for further parallel development.

8.2.2 Teams in the Conflict-Free Strategy

The decomposition of a model into packages is also used to dictate the structure of the team of developers working on the model. Each such team works on a distinct package of the model, i.e. for n packages we will have n teams working in parallel under the conflict-free strategy, each on one of these distinct packages. Packages can be hierarchical, i.e. a package can contain other packages. We have seen an example of this in Figure 8.2. We use this hierarchical structuring of a package to likewise structure the teams working on the model under the conflict-free strategy. Teams, in our approach, can be hierarchical and the hierarchical decomposition of a package naturally leads to the decomposition of the team working on the package into subteams.

Consider the hierarchical package P as sketched in Figure 8.4. It contains the subpackages $P_{1,1}$ and $P_{1,2}$ and each of these subpackages is further split up into two smaller subpackages ($P_{2,1}$ and $P_{2,2}$, and $P_{2,3}$ and $P_{2,4}$, respectively). A team T is working (exclusively) on package P, as indicated by the dotted

arrow from P to T. This team T is split up into two teams that work on the two subpackages of P, and one of these teams is further split up, as dictated by the package architecture. The conflict-free strategy is thus used to manage the efforts of T together with the other teams working on the other packages. The same strategy is also used within the hierarchical package P to internally structure the efforts of team T using subteams. Note that this is not required: we have not further split up team $T_{1,2}$ because we have chosen to keep one large team to work on the entire package $P_{1,2}$. The conflict-free strategy can thus be used to parallelize the development of the model into parts, up to the number of packages that exist in a model at the deepest level of nesting. The choice of packages then partially dictates the structure of the teams.

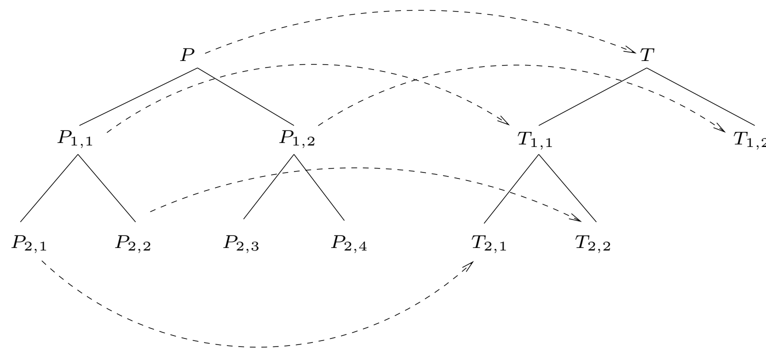

Fig. 8.4. Hierarchical teams.

Note that during a renegotiation phase the team structure is affected to reflect the new distribution of packages. In Figure 8.3, we (temporarily) merge the teams working on packages P_2 and P_3 in order to reflect the fact that they are now working together to determine the new interactions between these packages. Hence, the initial team structure is determined by the architecture of the initial model and is adapted dynamically due to renegotiation. In the example of Figure 8.3, the wayward packages P_2 and P_3, which are edited by the teams T_2 and T_3, respectively, are temporarily placed in a package New during renegotiation. These two teams together are then responsible for modifying this new package, as sketched in Figure 8.5.

The structure of the model and the structure of the teams are thus tightly coupled. The initial model determines how the teams can be distributed over the packages for parallel development. On the other hand, desired non-local changes of one of the teams can lead to a (temporary) change in architecture. The model itself is “actively” involved in the development process. This

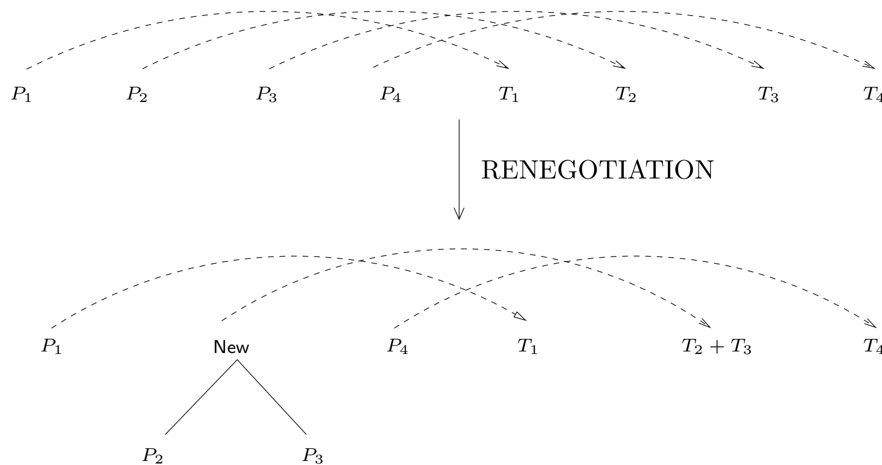

Fig. 8.5. Merging teams.

contrasts with many workflow or software process models, where the model under development is not really relevant (see, e.g., [KB95] and [DKW99]). They focus more on the documents to be produced, and their timing. The contents of these documents however do not play an explicit role.

In our approach, the activities of the teams can be divided into two categories: those which are internal to a team and those which involve other teams (due to renegotiation). The management of the teams in the conflict-free strategy can be divided along these lines. On the one hand, management can be localized and is only concerned with coordinating the changes to one package by one team. Here the focus is on coordinating a relatively small group in a well-defined context. On the other hand, the structure of the teams can be a separate management concern. The management of the hierarchical structure of the model and of the teams as given in Figure 8.4 can become an issue in its own right. This is a relatively more complex job than "just" managing one team. Seniority and experience can come into play when determining which role is played by which individual. Relatively unexperienced individuals should manage relatively small teams such as $T_{2,1}$, while a more experienced manager should lead the more complex team $T_{1,1}$. The most experienced manager can decide whether changes leading to renegotiation fit within the direction the model should be heading in order to match its specification.

Note that we do not discuss how teams should be led. We postulate a group of people who together perform a common editing of one package. We do not claim that they should coordinate their work in any specific way.

We just define the extent of their possible changes by only allowing local changes. We also do not discuss how two separate teams, when integrated, should coordinate their efforts. This is a nontrivial task, especially if the two teams previously worked according to different philosophies. We just constrain the extent of their possible actions as a new, larger team. This is a topic of research with strong sociological impact, which is however outside the scope of this thesis, but naturally fits well within CSCW. The conflict-free strategy does provide a context within which knowledge about how people work can be embedded.

8.2.3 Teams Modeled by Team Automata

We now sketch how a hierarchical team structure, as induced by the structure of the model under development in the way described in the previous subsections, can be modeled in terms of team automata. We interpret actions as operations or changes of (a package of) the model. Since internal actions of a component automaton cannot be observed by any other component automaton, these actions are ideally suited for representing a local change to a package using the conflict-free strategy. The external actions, on the other hand, are ideal for modeling the collaboration between packages.

In Figure 8.6 we represent our example teams T_2 and T_3 by two quite trivial component automata $\mathcal{T}_2$ and $\mathcal{T}_3$, respectively. The states of $\mathcal{T}_2$ are p_1, p_2, and p_3, whereas q_1 and q_2 are the states of $\mathcal{T}_3$. The wavy arcs indicate the initial states p_1 and q_1 of $\mathcal{T}_2$ and $\mathcal{T}_3$, respectively. $\mathcal{T}_2$ has no input actions, output actions a and d, and internal actions b and c, while $\mathcal{T}_3$ only has output actions, viz. a and d. Their transition relations are as depicted in Figure 8.6. Now a possible scenario could be as follows. First $\mathcal{T}_2$ and $\mathcal{T}_3$ execute output action a in parallel. Consequently $\mathcal{T}_2$ executes a number of internal actions (i.e. local changes to its package without consulting the other teams). Eventually both component automata can execute output action d in parallel, after which this procedure can be repeated. Naturally we could imagine also $\mathcal{T}_3$ having some internal actions (i.e. local changes) to execute once in a while.

Note that $\{\mathcal{T}_2, \mathcal{T}_3\}$ is a composable system. In Figure 8.7, the state-reduced version $(\mathcal{T}_{2,3})_S$ of a team automaton $\mathcal{T}_{2,3}$ over $\{\mathcal{T}_2, \mathcal{T}_3\}$ is given. Note that output actions a and d are *sopp* in $\mathcal{T}_{2,3}$, requiring both $\mathcal{T}_2$ and $\mathcal{T}_3$ to change state, whereas only $\mathcal{T}_2$ is changing state when internal actions b or c are executed. The behavior of both $\mathcal{T}_2$ and $\mathcal{T}_3$ is thus reflected in the behavior of $\mathcal{T}_{2,3}$. In our interpretation, such peer-to-peer types of synchronization can represent changes which affect two or more packages, i.e. non-local changes. The external actions of $\mathcal{T}_{2,3}$ thus represent the shared operations on the

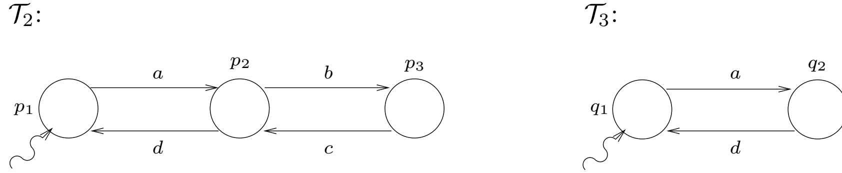

Fig. 8.6. Component automata $\mathcal{T}_2$ and $\mathcal{T}_3$.

merged packages P_2 and P_3. Note that we could also use master-slave types of synchronization to model boss-employee relations in which employees have to follow orders from their bosses.

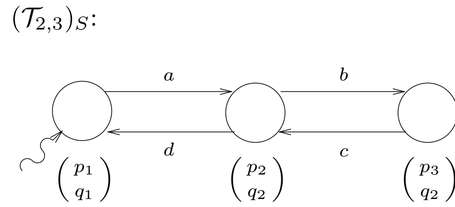

Fig. 8.7. State-reduced team automaton $(\mathcal{T}_{2,3})_S$ over $\{\mathcal{T}_2, \mathcal{T}_3\}$.

The external actions of $\mathcal{T}_{2,3}$ consequently can be hidden in order to obtain a team automaton with only internal actions, i.e. with only local operations on its packages. The resulting team automaton can then be used as a component automaton in a larger team automaton. In this way, subteams and hierarchical team structures can be modeled. In Figure 8.8, e.g., team automaton $\mathcal{T}$ is defined as a composition of team automaton $\mathcal{T}_{2,3}$ with certain component automata $\mathcal{T}_1$ and $\mathcal{T}_4$. As such, team automata are well suited for modeling (the actions of) the hierarchical teams in the conflict-free strategy.

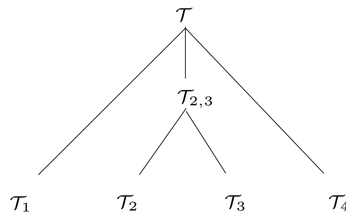

Fig. 8.8. A team automaton $\mathcal{T}$ over $\mathcal{T}_1$, $\mathcal{T}_{2,3}$, and $\mathcal{T}_4$.

8.2.4 Conclusion

In this section we have discussed a conflict-free strategy for the development of a model by several teams of developers working in parallel on distinct packages of the model. We guided the changes made by each team so as to ensure no conflicts occur during the merge of the produced efforts. This approach is scalable as each package can be developed in a similar fashion by splitting the package up further. We have moreover shown how packages under development can (temporarily) be merged during a renegotiation phase, if we need changes to a package that would invalidate the conflict-free strategy.

Additionally, we have discussed how the hierarchical structure of the model in packages can be used to structure the teams working on the model. The top-down decomposition of a model into packages guides the decomposition of the people working on the model into similarly structured teams. The renegotiation phase, when packages are temporarily merged, then gives heuristics on how the teams should further cooperate to implement changes without generating conflicts. We have sketched how this can formally be modeled by team automata.

The conflict-free strategy, along with the explicit discussion on the team structure and its actions, brings the worlds of CSCW, software engineering, software configuration management, and process modeling very close together. We have discussed how a large model can be developed and how the work between the people doing the actual work can be coordinated. Special to the approach is that the subject of the work, the model under development, is used to structure the work and thus plays an active part in deciding which changes are possible.

8.3 Spatial Access Control

As the complexity of reactive (computer) systems continues to increase, abstractions tend to be especially useful. For this reason, computer science often introduces and studies various models of computation that allow enhanced understanding and analysis. Computer science has also created a number of interesting metaphors (e.g., the desktop metaphor) that aid in end user understanding of computing phenomena. This section is concerned with a model and a metaphor. The model is team automata and the metaphor is *spatial access control*, which is based upon current notions of virtual reality, and helps demystify concepts of access control matrices and capability structures for the end user ([BB99]).

Our aim here is to connect the metaphor of spatial access control to the framework of team automata, and to show through examples how this combination facilitates the identification and unambiguous description of some key issues of access control. The rigorous setup of the framework of team automata allows one to formulate, verify, and analyze general and specific logical properties of various control mechanisms in a mathematically precise way. In realistically large (computer) systems, security is a big issue, and team automata allow formal proofs of correctness of its design. Moreover, a formal approach as provided by the team automata framework forces one to unambiguously describe control policies and it may suggest new approaches not seen otherwise. There is a large body of literature concerning topics like security, protection, and awareness in (computer) systems. Although team automata are potentially applicable also to these areas, we are currently not concerned with issues outside of spatial access control. We will conclude with a discussion of some variations and extensions of our setup.

We now begin by discussing the spatial access control metaphor by means of an example and subsequently we show how certain spatial access control mechanisms can be made precise and given a formal description using team automata. We first introduce information access modeling by granting and revoking access rights, and show how *immediate* versus *delayed revocation* can be formulated. Subsequently we extend our study to the more complex issue of meta access control and, finally, we show how team automata can deal with *deep* versus *shallow revocation*.

8.3.1 Access Control

A vital component of any (computer) system or environment is security and information *access control*, but this is sometimes done in a rather ad hoc or inadequate fashion with no underlying rigorous, formal model. In typical electronic file systems, access rights such as read-access and write-access are allocated to users on some basis such as “need to know”, ownership, or ad hoc lists of accessors. Within groupware systems, there are typically needs for more refined access rights, such as the right to scroll a document that is being synchronously edited by a group in real time. Furthermore, the granularity of access must sometimes be more fine grained and flexible, as within a software development team. Moreover, it is important to control access meta rights. For example, it may be useful for an author to grant another team member the right to grant document access to other non-team members (i.e. delegation). Various models have been proposed to meet such requirements (see, e.g., [SD92], [Rod96], and [Sik97]).

We use a spatial access metaphor based upon work of Bullock and colleagues in [BB97] and [BB99]. There, access control is governed by the rooms, or spaces, in which subjects and objects reside, and the ability of a subject to traverse space in order to get close to an object. Bullock also implemented a system called SPACE to test out some of these ideas ([Bul98]). A basic tenet of the SPACE access model is that a fundamental component of any collaborative environment is the environment itself (i.e. the space). It is the shared territory within which information is accessed and interaction takes place. Often this shared space is divided into numerous regions that segment the space. This allows decomposition of a very large space into smaller ones for manageability. It also allows cognitive differentiation (i.e. different concerns, memories, and thoughts associated with different regions), and distributed implementation (i.e. different servers for different regions).

By adopting a spatial approach to access control, the SPACE metaphor exploits a natural part of the environment, making it possible to hide explicit technical security mechanisms from end users through the natural spatial makeup of the environment. These users can then make use of their knowledge of the environment to understand the implicit security policies. Users can thus avoid understanding technical concepts such as so-called access matrices, which helps to avoid misunderstandings.

We consider here a virtual reality, in which a user can traverse from room to room by using keyboard keys, the mouse, or fancier devices. It is a natural and simple extension to assume that access control checking happens at the boundaries (doors) between spaces (rooms) when a user attempts to move from one room to another. If the access is OK, then the user can enter and use the resources associated with the newly entered room.

To illustrate the various concepts throughout this section, we present a simple running example which is concerned with read and write access to a file F by a user Kwaku. This file might be any data or document that is stored electronically within a typical file system. The file system keeps track of which users have which access rights to the file F. Three types of access rights are possible for a file F: null access (implying the user can neither read nor write the file), read access (implying the user cannot write the file), and full access (implying the user can read and write — i.e. edit — the file).

In security literature, authentication deals with verification that the user is truly the person represented, whereas authorization deals with validation that the user has access to the given resource. Assume that when Kwaku logs into the system, there is an authentication check. Then whenever he tries to read or write F, authorization checking occurs, and Kwaku is either allowed

the access, or not. Using the SPACE metaphor, the above three types of access rights can be associated with three rooms as shown in Figure 8.9.

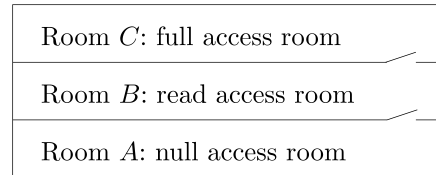

Fig. 8.9. A rooms metaphor for access control.

Room A is associated with no access to the document, room B is associated with read access, and room C models full access. Suppose Kwaku is in room B, the reading room. Presence in this room means that any time Kwaku decides to read F, he can do so. However, if he attempts to make changes to F, then he will fail because he does not have write access in room B. There are doors between rooms, implying that user access rights can be dynamically changed by changing rooms. We discuss this dynamic change in more detail later in this section.

This access mechanism satisfies a number of end user friendly properties: it is simple, understandable by non-computer people, relatively natural and unobtrusive, and elegant. Later we show how modeling this type of access metaphor via team automata adds precision, mathematical rigor, and analytic capabilities.

We now show how to model our access control example in the team automata framework. The component automaton $\mathcal{C}^C$ depicted in Figure 8.10(a) corresponds to room C of Figure 8.9, as it models full access to file F. The states of $\mathcal{C}^C$ are C_e modeling an empty room, C_n modeling F is not accessed, C_r modeling F is being read, and C_w modeling F is being written (edited). The wavy arc in Figure 8.10(a) denotes the initial state C_e. The actions of $\mathcal{C}^C$ are e_{BC} (enter room), e_{CB} (exit room), r^C (begin reading), $\underline{r}^C$ (end reading), w^C (begin writing), and $\underline{w}^C$ (end writing).

$\mathcal{C}^C$ thus has the transitions (C_e, e_{BC}, C_n), (C_n, e_{CB}, C_e), (C_n, r^C, C_r), $(C_r, \underline{r}^C, C_n)$, (C_r, w^C, C_w), and $(C_w, \underline{w}^C, C_r)$. Now transition (C_e, e_{BC}, C_n), e.g., shows that in $\mathcal{C}^C$ we can go from state C_e to C_n by executing action e_{BC}. We also see that transitioning directly from C_n to C_w is not possible. Furthermore, entering and exiting room C may only occur via state C_n. We choose to specify actions r^C, $\underline{r}^C$, w^C, and $\underline{w}^C$ as internal actions of $\mathcal{C}^C$, and e_{BC} and e_{CB} as external actions of $\mathcal{C}^C$. Both e_{BC} and e_{CB} clearly should be externally visible and therefore cannot be internal. For the moment we

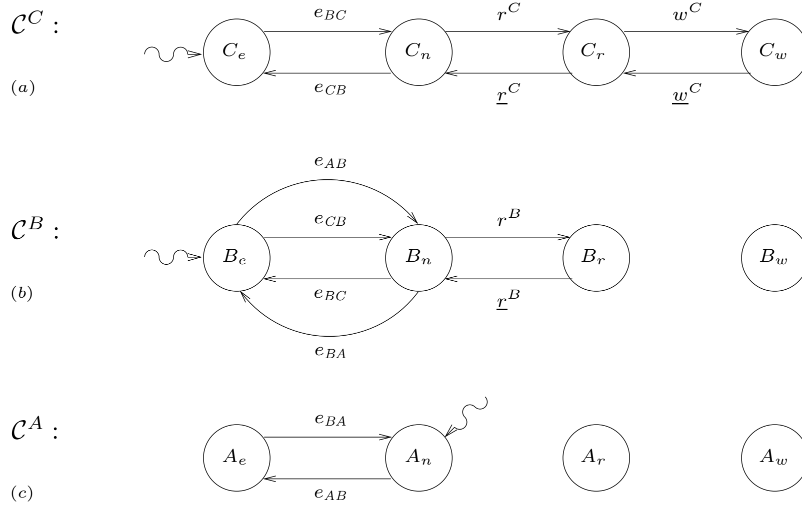

Fig. 8.10. Component automata $\mathcal{C}^C$, $\mathcal{C}^B$, and $\mathcal{C}^A$: rooms C, B, and A.

choose them to be output actions. These two external actions are candidates for being synchronized with actions of the same name in other component automata when forming a team automaton over $\mathcal{C}^C$ and the two component automata described next.

Component automata $\mathcal{C}^B$ and $\mathcal{C}^A$ corresponding to rooms B and A, respectively, are somewhat similar to $\mathcal{C}^C$. However, write access is denied in rooms B and A and read access is denied in room A. Component automata $\mathcal{C}^B$ and $\mathcal{C}^A$ are depicted in Figure 8.10(b,c). Note that $\mathcal{C}^A$ has initial state A_n (hence initially room A is not empty) and that both $\mathcal{C}^B$ and $\mathcal{C}^A$ have states unreachable from the initial state. Actions r^B and $\underline{r}^B$ are internal, while the rest of the actions of $\mathcal{C}^B$ and $\mathcal{C}^A$ are external (output) actions.

Now we want to combine $\mathcal{C}^C$, $\mathcal{C}^B$, and $\mathcal{C}^A$ into one team automaton reflecting a given access policy. They clearly form a composable system $\{\mathcal{C}^C, \mathcal{C}^B, \mathcal{C}^A\}$ and we combine them into a team automaton $\mathcal{T}^{CBA}$ as follows. Since each state of $\mathcal{T}^{CBA}$ is a combination of a state from $\mathcal{C}^C$, a state from $\mathcal{C}^B$, and a state from $\mathcal{C}^A$, $\mathcal{T}^{CBA}$ has $4^3 = 64$ states. Initially $\mathcal{T}^{CBA}$ is in state (A_n, B_e, C_e), which means one starts in room A, while rooms B and C are empty.

Assuming that one can have only one kind of access rights at a time, two of the rooms should be empty at any moment in time. This means that $\mathcal{T}^{CBA}$ should be defined in such a way that in each of its reachable states two of the three component automata are always in state "empty". We let the component automata synchronize on the external actions e_{AB}, e_{BA}, e_{BC},

and e_{CB}. Each such synchronized external action of $\mathcal{T}^{CBA}$ corresponds to exiting a room while entering another. Synchronization of action e_{AB}, e.g., models a move from room A to room B. This move is represented by the transition $((A_n, B_e, C_e), e_{AB}, (A_e, B_n, C_e))$ showing that in component automaton $\mathcal{C}^A$ we exit room A, in automaton $\mathcal{C}^B$ we enter room B, and in component automaton $\mathcal{C}^C$ we do nothing (i.e. remain idle). This represents a change in access rights from null access (in room A) to read access (in room B). We do not include, e.g., the transition $((A_n, B_e, C_e), e_{AB}, (A_e, B_e, C_e))$ which would let the user exit room A but never enter room B. Furthermore, the user could be in more than one room at a time if we would allow transitions like $((A_n, B_e, C_e), e_{AB}, (A_n, B_n, C_e))$. In T^{CBA} we include only the four transitions representing the synchronized changing of rooms. In each of these transitions, one component automaton is idle. Since all internal (read and write related) actions are maintained, in each of these only that component automaton is involved to which such an action belongs.

The state-reduced version $\mathcal{T}_S^{CBA}$ of the thus defined team automaton $\mathcal{T}^{CBA}$ over $\{\mathcal{C}^C, \mathcal{C}^B, \mathcal{C}^A\}$ is depicted in Figure 8.11.

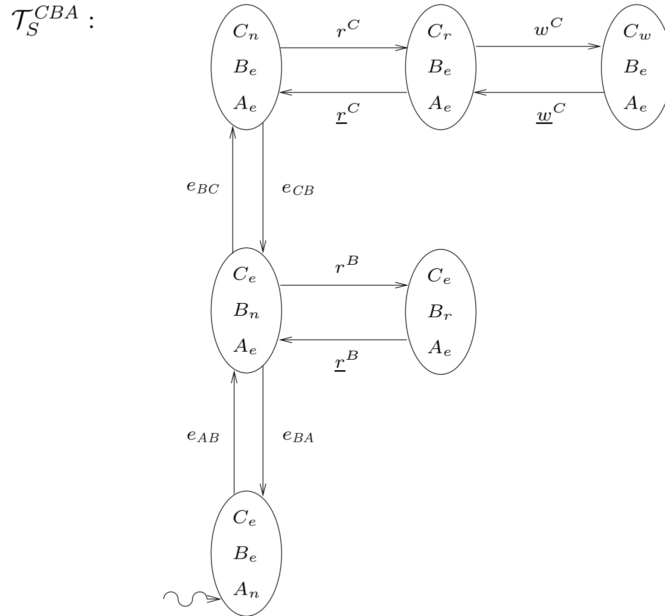

Fig. 8.11. State-reduced team automaton $\mathcal{T}_S^{CBA}$ over $\{\mathcal{C}^C, \mathcal{C}^B, \mathcal{C}^A\}$.

Recall that $\mathcal{T}^{CBA}$ is not the only team automaton over $\{\mathcal{C}^C, \mathcal{C}^B, \mathcal{C}^A\}$. Also recall that the decision to consider e_{AB}, e_{BA}, e_{BC}, and e_{CB} as output actions in all component automata of $\mathcal{T}^{CBA}$ was made more or less arbitrarily. In fact, it depends on how one views the action of entering and exiting a room within the team automaton $\mathcal{T}^{CBA}$. By choosing all of those actions to be output (and thus of the same type), exiting one room and entering another is seen as a *sopp* action. Recall that, on the other hand, master-slave types of synchronization occur when input actions can only occur as a response (slave) to output actions. In our example, assume that one views the changing of rooms as an action initiated by leaving a room and forcing the room that is entered to accept the entrance. Then one would name, e.g., e_{AB} an output action of $\mathcal{C}^A$ and an input action of $\mathcal{C}^B$, and e_{BA} an output action of $\mathcal{C}^B$ and an input action of $\mathcal{C}^A$. This causes both e_{AB} (with master $\mathcal{C}^A$ and slave $\mathcal{C}^B$) and e_{BA} (with master $\mathcal{C}^B$ and slave $\mathcal{C}^A$) to be *sms*. Likewise for the other actions.

In addition, Section 5.4 defines strategies that lead specifically to uniquely defined combinations of peer-to-peer and master-slave types of synchronization within team automata. The team automata framework allows one to model many other features useful in virtual reality environments. A door, e.g., can be extended to join more than two rooms since any number of component automata can participate in an output action. Furthermore, as said before, a user could be in more than one room at a time.

8.3.2 Authorization and Revocation

We continue our running example by adding Kwaku, a user whose access rights to file F will be checked by the access control system $\mathcal{T}^{CBA}$. Kwaku is represented by component automaton $\mathcal{C}^U$, depicted in Figure 8.12. This extension complicates our example in the sense that Kwaku's read and write access rights can be changed independently of his whereabouts. Only to enter a room he has to be authorized. Thus access rights are no longer equivalent with being in a room, but rather with the possibility to enter a room. To add this to the team automaton formalization, we will use the feature of iteratively constructing team automata with team automata as their constituting component automata.

Kwaku starts in state U_n with no access rights. The actions $m(r)$, $\underline{m}(r)$, $m(w)$, and $\underline{m}(w)$ model the (meta) operations of "being granted read access", "being revoked read access", "being granted write access", and "being revoked write access", respectively. Since these clearly are passive actions from Kwaku's point of view, we choose all of them to be input actions. Note that Kwaku can end up in state U_w if and only if he was granted access rights

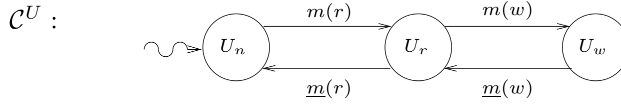

Fig. 8.12. Component automaton $\mathcal{C}^U$: user Kwaku.

to read and to write, i.e. actions $m(r)$ and $m(w)$ have taken place. When Kwaku's write access is consequently revoked by transition $(U_w, \underline{m}(w), U_r)$, he ends up in state U_r.

Now suppose that we want to model Kwaku's options for editing file F, which is protected by the access control system $\mathcal{T}^{CBA}$. Then we would like to compose a team automaton over $\mathcal{T}^{CBA}$ and $\mathcal{C}^U$. To do so, first note that $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$ is a composable system. Next we choose a transition relation, i.e. for each action a subset from its complete transition space in $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$ is selected, thereby formally fixing an access control policy for Kwaku under the constraints imposed by $\mathcal{T}^{CBA}$.

The initial state of any team over $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$ is (A_n, B_e, C_e, U_n), i.e. Kwaku is not yet editing F and is in the virtual room A without access rights. Now imagine the access rights to be keys. Hence Kwaku needs the right key to enter reading room B, i.e. action $m(r)$ must take place before action e_{AB} becomes enabled. This action $m(r)$ leads us from the initial state to (A_n, B_e, C_e, U_r). Now Kwaku has the key to enter room B by $((A_n, B_e, C_e, U_r), e_{AB}, (A_e, B_n, C_e, U_r))$. This transition models the acceptance of Kwaku's entrance of room B, i.e. this action is the authorization activity mentioned earlier. Hence our choice of the transition relation fixes the way we deal with authorization. If we would include, e.g., $((A_n, B_e, C_e, U_n), e_{AB}, (A_e, B_n, C_e, U_n))$ in the transition relation, this would mean that Kwaku can enter room B without having read access rights for F. Note however that since transitions involving internal actions of either $\mathcal{T}^{CBA}$ or $\mathcal{C}^U$ by definition cannot be pre-empted in any team over $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$, our transition relation must contain $((A_e, B_n, C_e, U_n), r^B, (A_e, B_r, C_e, U_n))$. Hence Kwaku, once in room B, can always begin reading file F. By not including $((A_n, B_e, C_e, U_n), e_{AB}, (A_e, B_n, C_e, U_n))$ in our transition relation we avoid that Kwaku can read F without ever having been granted read access. This leads to the question of the revocation of access rights.

As argued, (A_e, B_n, C_e, U_r) — meaning that Kwaku is in room B with reading rights — will be a reachable state. Now imagine that while in this state Kwaku's reading rights are revoked by $\underline{m}(r)$. To which state should this action lead, i.e. in what way do we handle revocation of access rights? We could opt for modeling *immediate revocation* or *delayed revocation*. The

latter is what we have chosen to model first. Thus our answer to the question above is to include $((A_e, B_n, C_e, U_r), \underline{m}(r), (A_e, B_n, C_e, U_n))$. The result is that Kwaku can pursue his activities in room B, but cannot re-enter the room once he has left it (unless his read access has been restored). He is thus still able to read (browse) F, but the moment he decides to re-open the file this fails. Likewise, if Kwaku is writing F when his writing right is revoked, then he can continue editing (typing in) F, but he cannot re-enter room C as long as his write access right has not been restored. On this side of the revocation spectrum, a user can thus continue his or her current activity even when his or her rights have been revoked. He or she can do so until he or she wants to restart this activity, at which moment an authorization check is done to decide if he or she has the right to restart this activity. In some applications, this may be an intolerable delay.

Immediate revocation, on the other hand, means the following. If a user is reading when his or her reading right is revoked, then the file immediately disappears from view, while if a user is writing when his or her writing right is revoked, then the edit is interrupted and writing is terminated in the middle of the current activity. In some applications, this is overly disruptive and unfriendly. If we would want to incorporate immediate revocation into our example we would have to adapt our distribution of actions a bit. As said before, since r^B is an internal action we cannot disallow action r^B to take place after $((A_e, B_n, C_e, U_r), \underline{m}(r), (A_e, B_n, C_e, U_n))$ has revoked Kwaku's reading rights. If we instead choose r^B to be an external action, we are given the freedom not to include $((A_e, B_n, C_e, U_n), r^B, (A_e, B_r, C_e, U_n))$ in our transition relation. The result is that as long as Kwaku is not being granted read access by action $m(r)$, the only way left to proceed for Kwaku in state (A_e, B_n, C_e, U_n) is to exit room B by $((A_e, B_n, C_e, U_n), e_{BA}, (A_n, B_e, C_e, U_n))$. Modeling immediate revocation thus requires that actions such as r^B are visible, since in that way we can choose them not to be enabled in certain states. Immediate revocation also implies that we still want Kwaku to be able to stop reading and leave state (A_e, B_r, C_e, U_n) by $((A_e, B_r, C_e, U_n), \underline{r}^B, (A_e, B_n, C_e, U_n))$. Action $\underline{r}^B$ can thus remain internal.

This finishes the description of a part of a team automaton $\mathcal{T}$ over $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$. In Figure 8.13 the state-reduced version $\mathcal{T}_S$ of $\mathcal{T}$ (for delayed revocation) is depicted.

Recall that team automata are intended to be used to model (logical) design issues. An action can take place provided certain preconditions hold, and affects only states of those component automata involved in that action. Hence at this level there is no notion of time and no means are provided to give one action priority over another. A result of the lack of a notion of time

Fig. 8.13. Team automaton $\mathcal{T}_S$ over $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$.

is, e.g., that nothing can be said about how long it takes before Kwaku has left reading room B after his reading access right has been revoked. However, time and priorities may be added to the basic model as extra features.

Again, $\mathcal{T}$ is not the unique team automaton over $\{\mathcal{T}^{CBA}, \mathcal{C}^U\}$, but it is a team automaton one obtains by choosing a specific transition relation with a specific protocol in mind. Once again this shows that the freedom of the team automata model to choose transition relations offers the flexibility to distinguish even the smallest nuances in the meaning of one's design. Another interesting feature of the team automata framework is shown by the

following application of the results proven in Section 5.2 to our running example. In whatever order one chooses to construct a team automaton over the component automata $\mathcal{C}^C$, $\mathcal{C}^B$, $\mathcal{C}^A$, and $\mathcal{C}^U$, we know that it will always be possible to construct the team $\mathcal{T}$ discussed above. This means that instead of first constructing $\mathcal{T}^{CBA}$ over $\{\mathcal{C}^C, \mathcal{C}^B, \mathcal{C}^A\}$, and then adding $\mathcal{C}^U$, we could just as well have constructed an iterated team by, e.g., starting from the user component automaton $\mathcal{C}^U$ and adding successively the component automata $\mathcal{C}^C$, $\mathcal{C}^B$, and $\mathcal{C}^A$ modeling the access rights that can be exercised. Moreover, independent of the way a team automaton over $\mathcal{C}^C$, $\mathcal{C}^B$, $\mathcal{C}^A$, and $\mathcal{C}^U$ is constructed, more component automata can be added.

As an example, suppose that Kwaku has other interests than the file F. Hence imagine a component automaton $\mathcal{T}^{NBA}$ in which he can transition into a state in which he plays some basketball. Then we may construct a team over the team automaton $\mathcal{T}$ just described and the component automaton $\mathcal{T}^{NBA}$ modeling when Kwaku is entitled — or perhaps even forced — to have a break (which is of some importance in these times of RSI). In general, new component automata can be added to a given team automaton at any moment of time, without affecting the possibilities of any new additions. We thus conclude once again that the team automata framework scores high on scalability. We will come back to this shortly.

8.3.3 Meta Access Control

Until now we have seen how team automata can be used to describe the control of a user's access to a file depending on his or her rights. Here we further elaborate on the granting and revoking of access rights and we consider *meta access control*. This means that privileges such as granting and revoking of rights can themselves be granted and revoked. The complicated (recursive) situations that may arise in this fashion depend on the chosen (meta) access control policy and we demonstrate how they can unambiguously and concisely be defined in terms of team automata.

Figure 8.14 shows a component automaton $\mathcal{C}^0$ that models a building with three levels — A, B, and C — corresponding to null access, read access, and full access, respectively. This component automaton shows the same access structure as the three rooms of Figure 8.10. Now, however, the status of the user directly determines the level he or she operates on and the granting and revoking of access rights is identified with changing levels. This differs from the previous example where the status of the user only determined his or her rights to enter a room.

Consequently, in $\mathcal{C}^0$ the user moves in two dimensions: vertically between levels A, B, and C — indicating the dynamic change in access rights Kwaku

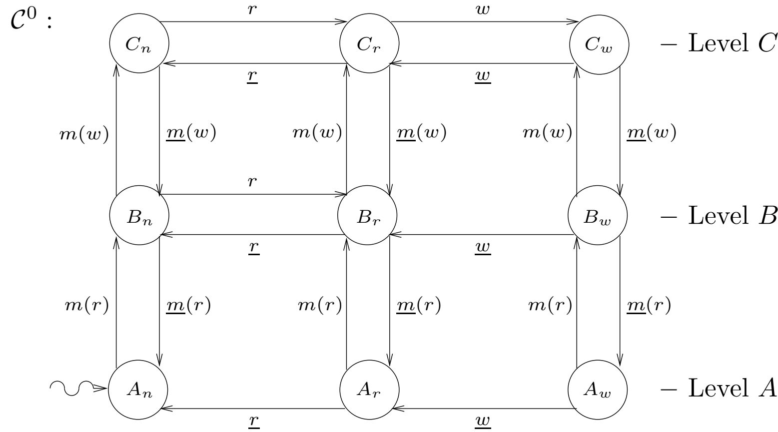

Fig. 8.14. Component automaton $\mathcal{C}^0$: the access building.

has for F — and horizontally between the states "null", "reading", and "writing" — indicating the current activities of Kwaku with respect to F. Notice that in $\mathcal{C}^0$, e.g., the state B_w meaning that Kwaku is writing while having read access but no write access, can only be reached from C_w by an action $\underline{m}(w)$ or from A_w by an action $m(r)$. Hence this state B_w can be entered only when Kwaku is writing while his status changes. There is no transition to B_w at level B. A similar remark holds for states A_r and A_w, which can be entered only from level B by the read access revocation action $\underline{m}(r)$. States such as A_r, A_w, and B_w are called *irregular states* because they are not reachable at their own level.

To model meta access control, we assume the existence of a system administrator, Abena, who can change Kwaku's rights. Hence Abena has the right to grant and revoke access by Kwaku to F. For this reason we have chosen all actions of granting and revoking access rights in $\mathcal{C}^0$ to be input actions, while all actions of reading and writing are output actions. The right to grant and revoke are legitimate rights, but they are not directly applied to F. They are in fact meta operations — hence $m(r)$ and $m(w)$ — and the rights to apply these meta operations are meta rights. Similarly, if there is a creator, Kwesi, who can allow (and disallow) Abena to grant and revoke, then Kwesi has meta meta rights. Kwesi has the meta meta right to grant and revoke Abena's meta rights to grant and revoke Kwaku's access rights to F. A typical action of Kwesi is $\underline{m}^2(w)$, which revokes Abena's right to grant and revoke write access to Kwaku.

The notion of meta clearly extends to arbitrary layers. An example of such a multi-layered structure of meta can be seen in the journal refereeing process. The creator of a document may delegate publication responsibilities to co-authors who may select a journal and grant $m^2(r)$ rights to the editor-in-chief. The editor-in-chief may grant $m(r)$ rights to assistant editors who can then grant and revoke read access to reviewers. An interesting question now arises as to the effect of revocation: should revocation of a meta right also revoke the rights that were passed on to others? This is the issue of *shallow revocation* versus *deep revocation*. Shallow revocation means that a revoke action does not revoke any of the rights that were previously passed on to others, whereas deep revocation means that a revoke action does revoke all rights previously passed on. Team automata can be used to model shallow, deep, or even hybrid revocation. Shallow revocation is often the easiest to model, whereas deep revocation is known as a big challenge to model and implement ([DS98]). We now show how deep revocation can be modeled using team automata.

Figure 8.15 shows a component automaton capturing one layer (layer k) of a multi-layer meta access specification for our example of read and write access. We have already seen layer 0, viz. component automaton $\mathcal{C}^0$. For each value of $k \geq 1$ there are corresponding component automata that are directly related to layer k (viz. $\mathcal{C}^{k-1}$ at layer $k-1$ and $\mathcal{C}^{k+1}$ at layer $k+1$). For each such component automaton $\mathcal{C}^k$, the horizontal actions $m^k(r)$, $\underline{m}^k(r)$, $m^k(w)$, and $\underline{m}^k(w)$ are output actions, whereas the vertical actions $m^{k+1}(r)$, $\underline{m}^{k+1}(r)$, $m^{k+1}(w)$, and $\underline{m}^{k+1}(w)$ are input actions. For $k = 0$ we identify r with $m^0(r)$, $\underline{r}$ with $\underline{m}^0(r)$, w with $m^0(w)$, and $\underline{w}$ with $\underline{m}^0(w)$. Similarly, $m(r) = m^1(r)$, $\underline{m}(r) = \underline{m}^1(r)$, $m(w) = m^1(w)$, and $\underline{m}(w) = \underline{m}^1(w)$.

We can now define a multi-layered structure by recursively composing a team automaton over $\mathcal{C}^0$, $\mathcal{C}^1$, $\ldots$, and $\mathcal{C}^n$, for some $n \geq k$. Note that $\{\mathcal{C}^0, \mathcal{C}^1, \ldots, \mathcal{C}^n\}$ is a composable system. As mentioned before we can also build this team automaton in an iterated way starting from, e.g., a team over any two component automata $\mathcal{C}^k$ and $\mathcal{C}^{k+1}$. In Figure 8.16, the state-reduced version $(\mathcal{T}^k_{k-1})_S$ of a team automaton $\mathcal{T}^k_{k-1}$ over $\mathcal{C}^{k-1}$ and $\mathcal{C}^k$, representing layer $k-1$ and layer k of this layered structure, is depicted.

The transition relation of this team $\mathcal{T}^k_{k-1}$ is chosen with the modeling of deep revocation in mind. Finally, note that in Figure 8.16 we have added superscripts to distinguish the states in $\mathcal{C}^k$ from the states in $\mathcal{C}^{k-1}$, e.g., state B_r of $\mathcal{C}^k$ from state B_r of $\mathcal{C}^{k-1}$.

In our example, $\mathcal{C}^2$ represents the actions of the supervisor Kwesi and $\mathcal{C}^1$ those of Abena. Now consider Kwesi in state B^2_r. Then Figure 8.16 tells us that Abena must be in one of the three states B^1_n, B^1_r, or B^1_w. Assume

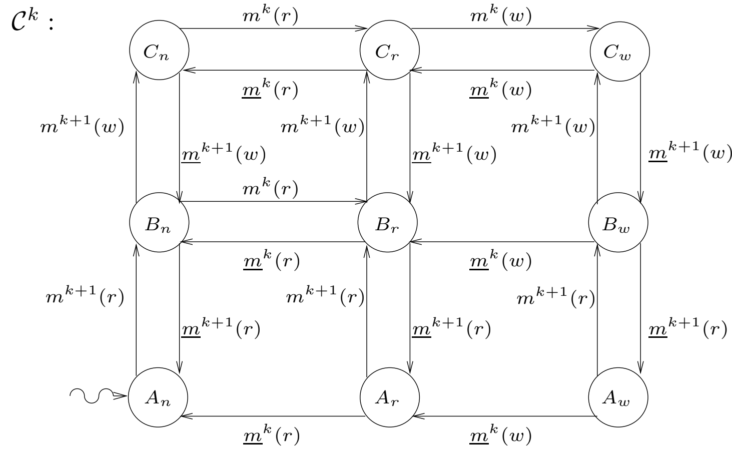

Fig. 8.15. Component automaton $\mathcal{C}^k$: meta access at layer k.

that Kwesi reached this state B_r^2 by performing action $m^2(r)$ from B_n^2, while Abena was in state A_n^1 having no rights to grant and revoke reading rights. Action $m^2(r)$ is an output action of $\mathcal{C}^2$ and an input action of $\mathcal{C}^1$, and our transition relation forces $\mathcal{C}^1$ to transition from A_n^1 to B_n^1. The interpretation is that Kwesi granted Abena the right to do read grants and revokes (to user Kwaku for file F).

Similarly, component automaton $\mathcal{C}^k$ can revoke the right to grant and to revoke read access from $\mathcal{C}^{k-1}$ at any time by performing output action $\underline{m}^k(r)$, and thus forcing $\mathcal{C}^{k-1}$ to perform this action — this time as an input action — as well. Continuing our example, this means that while in state B_r^2, Kwesi's read granting right may be revoked by action $\underline{m}^3(r)$ at any time. If this happens, Kwesi is forced into the irregular state A_r^2, which has only one possible output action, viz. $\underline{m}^2(r)$, leading to A_n^2. Whenever that action $\underline{m}^2(r)$ occurs it revokes Abena's right to change Kwaku's read access.

We thus observe two general rules of activity in such a team automaton over $\{\mathcal{C}^0, \mathcal{C}^1, \ldots, \mathcal{C}^n\}$, with each component automaton of the form depicted in Figure 8.15. First, when a "master" component automaton $\mathcal{C}^k$ where $1 \leq k \leq n$, transitions right (grant) or left (revoke), then the "slave" component automaton $\mathcal{C}^{k-1}$ must transition upward (gaining some access right) or downward (losing some access right). Secondly, the slave $\mathcal{C}^{k-1}$ may be forced to transition downward into an irregular state, in which case it will eventually transition to the left. $\mathcal{C}^{k-1}$ is itself a master and thus this transition to the left again forces a downward transition of $\mathcal{C}^{k-2}$, and so on until $\mathcal{C}^0$ on layer 0. Hence, as promised, we indeed model deep revocation.

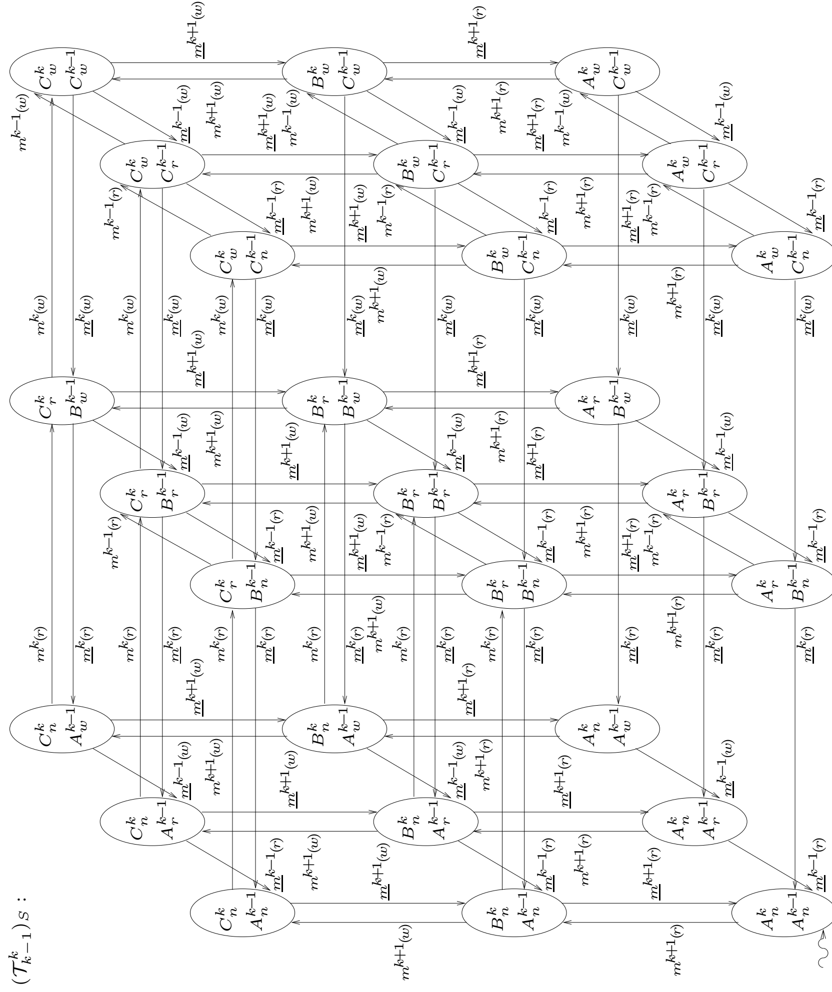

Fig. 8.16. State-reduced team automaton $(\mathcal{T}^k_{k-1})s$ over $\mathcal{C}^{k-1}$ and $\mathcal{C}^k$.

8.3.4 Conclusion

In this section we have demonstrated by means of examples how team automata can be used for modeling access control mechanisms presented through the metaphor of spatial access. The combination of the formal framework of team automata and the spatial access metaphor leads to a powerful abstraction well suited for a precise description of (at least some of the) key issues of access control. The team automata framework supports the design of distributed systems and protocols, by making explicit the role of actions and the choice of transitions governing the communication, coordination, cooperation, and collaboration. Examples include, e.g., peer-to-peer and master-slave types of synchronization, or heterogenous combinations thereof. Moreover, the formal setup and the possibility of a modular design provide analytic tools for the verification of desired properties of complex (computer) systems. Team automata are thus a fitting companion to the virtual spaces metaphor used in virtual reality systems that supports notions of rooms and buildings. Each space is represented by a component automaton, dynamic access changes are represented by joint external actions, while resource accesses within a space can be represented by internal actions.

Obviously there are numerous other possible examples as well as variations of the example we have considered above. For one, the assumption that write access can only be granted if read access has been granted can easily be dropped. Similarly, grant and revoke rights can be coupled more loosely. Read and write operations are specified here at the file level, but could also have been specified at the page level, object level, or record level, to name but a few. This might mean that delayed revocation is precisely the right choice. At the file level, the r and $\underline{r}$ actions might be seen at the user interface as open and close file. The w and $\underline{w}$ actions might be edit and save operations. When dealing with a transaction system, combinations of these operations might correspond to begin transaction and end transaction.

The team automata framework handles group decision making well and therefore allows convenient implementations of *distributed access control*. Distributed access control means that the supervisory work of granting and revoking access rights is administered by multiple agents. Thus Kwaku could have two administrative supervisors who must agree on any change of access rights. This can be modeled as an action of two masters and one slave: the actions would be output for both supervisors, requiring both to participate, and input for the slave. Alternatively, by including transitions with one supervisor being inactive, we can model the case of approval being required by either one of the two supervisors. Hybrids between pure master-slave and pure peer-to-peer types of synchronization, as in heterogenous team automata, are also

useful. All these variations are due to the fact that the choice of a transition relation is the crucial modeling issue of the team automata framework.

Recall that team automata model the logical architecture of a design. They abstract from concrete data, configurations, and actions, and only describe behavior in terms of a state-action diagram (structure), the role of actions (input, output, or internal), and synchronizations (shared actions). It is not feasible (nor necessary) to have a distinct component automaton for each individual, and for each file in an organization. In many situations, categories and roles are used rather than individuals. Any implementation would have the team automaton as a class entity, and an activation record for each person, containing their current state. Similarly, by keeping a status of the files one can model the criterion "only one person can write a file at a time, but many readers is OK". The model cast in the spirit of component automata depicting roles rather than individuals becomes much more useful and general, and avoids some notational problems of exponential growth.

As observed earlier, time and priorities are not incorporated in neither the spatial access metaphor nor the team automata model as discussed here. However, similar to the Petri net model one may consider to extend team automata with time and priorities (see, e.g., [ABC$^+$95], which focuses on performance analysis). When time and/or priorities are part of access control this would allow the designer to control the sojourn times in the local states and to control the resolution of conflicting actions.

Using team automata for modeling (spatial) access control forces one to make explicit and unambiguous design choices and at the same time provides the possibility of mathematically precise analysis tools for proving crucial design properties, without first having to implement one's design.