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

Beek, M.H. ter

9. Discussion

In this chapter we summarize the main contributions of this thesis and point out some topics worth further investigation. We moreover indicate how — in theory — team automata can be used for system design and where — in practice — they have actually been used.

Contributions of the Thesis

In this thesis we have formally presented team automata as a model for component-based system design. Team automata are based on the well-known method for modeling collaboration between system components by synchronizations of actions or transitions. A distinguishing feature of team automata is the freedom to choose on which actions and when their constituting component automata synchronize. In addition, there is the distinction of a team automaton's alphabet into input, output, and internal actions.

Through the classification of a broad range of ways to synchronize actions in team automata, a systematic study of the role that synchronizations play when modeling collaboration between system components has been conducted. To begin with, we have studied their effect on the inheritance of various automata-theoretic properties from team automata to their constituting component automata and subteams, and vice versa. We have furthermore studied their effect on the inheritance of various automata-theoretic properties from team automata to their constituting component automata and subteams, and vice versa. These studies are not complete and thus offer interesting pointers for further investigation.

The relation between team automata and two related models, viz. I/O automata and Petri nets, has been investigated in considerable detail. This has shown that I/O automata fit into the framework of team automata, whereas so-called non-state-sharing vector team automata can be translated into ITNCs — a model of vector-labeled Petri nets. Vector team automata are team automata in which the (team) actions have been replaced by vectors of (component) actions, from which the participation of a component automaton

in a synchronization can thus be seen immediately. Consequently, non-state-sharing vector team automata are the subclass of vector team automata with the characteristic that whether or not a synchronization can take place only depends on the local states of the component automata actively involved in that synchronization. As a result, synchronizations involving disjoint sets of component automata are independent, which would thus allow a concurrent semantics for non-state-sharing vector team automata. This is a point worth further investigation.

Team automata are naturally suited for component-based system design due to the fact that they can themselves be used as component automata of higher-level team automata. This allows the iterative composition of team automata. We have been able to show that iterated composition does not lead to an increase of the number of possibilities for synchronization. Every iterated team automaton over a composable system can be interpreted as a team automaton over that composable system, by reordering its state space and transition space. We have moreover been able to show that every team automaton can be iteratively composed over its subteams.

We have studied the computations and behavior of team automata in relation to those of their constituting component automata. Several types of team automata that satisfy compositionality could be identified. To describe the compositionality of team automata, we have had to develop an extensive theory of (synchronized) shuffles. An examiniation of the compositionality of further types of team automata is certainly a topic worth further investigation. This might very well require the introduction and analysis of more sophisticated types of shuffles.

Using Team Automata

Modeling a system as a team automaton in the early phases of design forces one to identify the active components of the system and to consider the intended communications and synchronizations in detail, which is bound to lead to a better understanding of system functionality and to explicit and unambiguous design choices. This forms the basis of further design and implementation, while at the same time the mathematically rigorous definitions provide the possibility of formal analysis tools for proving crucial design properties, without first having to implement the design.

In Theory

To model a system as a team automaton, first the components have to be identified. Each of them should be given a description in the form of an au-

tomaton — an easy to understand model that moreover forms the basis for system descriptions in a number of model-checking tools (see, e.g., [Hol91], [Kur94], [Hol97], and [Hol03]). Based on the idea of synchronizations of common actions, these components can be connected in order to collaborate. Within each component, a distinction has to be made between internal actions — which are not available for synchronization with other components — and external actions — which can be used to synchronize components and may be subject to synchronization restrictions. By assigning such different roles to actions it is possible to describe many types of collaboration.

Consequently, for each external action separately, a decision is made as to how and when the components should synchronize on this action. If the action is supposed to be a passive action that may not be under the component's local control, then it can be designated as an input action of that component, otherwise as an output action. If such a distinction between the roles of an external action is not necessary, then the choice is arbitrary. A natural option would be to make it an output action in all components in which it occurs. Once the synchronization constraints for each external action have been determined, one may apply, e.g., a maximality principle to construct a unique team automaton satisfying all constraints.

The team automata framework thus supports component-based system design by making explicit the role of actions and the choice of transitions that govern the collaboration between components. The crucial feature is the freedom of choice for the synchronizations collected in the transition relation of a team automaton. This is indeed one of the main reasons given in [Ell97] for introducing team automata to model groupware systems rather than using I/O automata for that purpose. Another important reason is that, in order for a team automaton to be capable of modeling various types of collaboration between its components by synchronizations of common actions, synchronizations between output actions of its components should not be excluded a priori. As a matter of fact, the peer-to-peer types of synchronization explicitly use the possibility to synchronize on output actions. Finally, no matter how convenient input enabling may be when modeling reactive systems, it does hinder a realistic modeling of collaborations that involve humans — in fact, Tuttle himself was the first to acknowledge this when he introduced I/O automata in [Tut87] (cf. Section 7.1) — while modeling such collaborations was one of the main reasons for the introduction of team automata.

In Practice

An increasing number of papers bears witness to the usefulness of team automata in the early design phase of reactive systems in general, and of

groupware systems in particular. Moreover, these examples are not limited to modeling within CSCW (see, e.g., [Ell97], [EK00], [Lav00], [BEKR01a], [BEKR01b], and [BB03]) but extend to areas such as software engineering (see, e.g., [HB00], [Hoe01], and [EG02]) and — most recently — security (see, e.g., [BLP03]). In fact, a spectrum from hardware components to protocols for interacting groups of people has been modeled by team automata. There is still quite some work left to do, though. For one, the components of a team currently cannot exchange any information, i.e. they have no private memory. In order to be useful also in later stages of the design of groupware systems (or to model, e.g., workflow systems) team automata should thus — among other things — be extended with the flow of information between components. An initial attempt in this direction was recently undertaken in [BCM03]. Furthermore, team automata are currently inappropriate for capturing aspects of group activity such as social aspects and informal unstructured activity.

We now close this Discussion with an initial observation on the potential of team automata within a process model recently introduced in the field of CSCW. In [Dew01], Dewan claims that traditional software process models such as the waterfall model and the spiral model — while efficient for describing the different phases in the life cycle of software in general — lack too many "collaboration-specific details" to be efficient for "collaborative systems". These are software systems including "both general infrastructures and specific applications for supporting collaboration". Therefore, Dewan proposes a new process model well suited for collaborative systems.

The initial phase of Dewan's model consists of decomposing the functionality of collaborative systems into smaller subfunctions, which can be worked upon more-or-less independently. Examples of such collaboration functions are listed in [DCS94] and [Dew01]. Among them are *merging* and *access control*. Merging combines independent versions into a single object, whereas access control determines the operations a user is authorized to perform. In Section 8.2 we showed how team automata could be applied — in a conflict-free strategy — to merge previously distributed packages back together. In Section 8.3 we consequently showed how access control mechanisms could be made precise and given a formal description using team automata. Team automata thus seem promising for modeling these two subfunctions of Dewan's process model.