Cooperative Concurrency Control for Design Environments

Ansgar Bredenfeld

GMD-SET
Schloß Birlinghoven
D-53754 Sankt Augustin, Germany

Abstract

In this paper, we present a new model for concurrency control that supports cooperation of design tools and designers in a design environment. We capture characteristic access and cooperation behaviour of design tools by activity types to guide a concurrency control component in access synchronization, conflict handling and inter-tool communication. Activity types allow to characterize cooperative work situations more precisely. This allows to improve designers awareness of conflicts in an environment of concurrently running design tools and supports them in interactive conflict resolution.

1 Introduction

Design environments consist of various design tools and a framework [1], [2] providing an infrastructure supporting design methodology management and design data management. An important service to be offered by a framework is the support of cooperation between designers as well as design tools. Design methodology management [3] deals with the organization of the design process. It provides means for project management related aspects of cooperation like team work and task assignment to designers. This information is represented by meta data which is used to control the execution of design flows.

Design flow engines usually operate on meta data and do not take the corresponding fine-grained design data into account. Therefore, they are of limited use to support tight cooperation of design tools sharing common design data. Nevertheless, this support is essential for tool interoperability. Since deep submicron design needs data-centered synchronization of tools for synthesis, floorplanning, placement and routing, tool interoperability will become a more important issue in the future. Therefore, cooperative work of design tools has to be adequately supported by a concurrency control component of a framework.

This research was supported in part by the commission of the EU under ESPRIT contract 7364

Cooperative work situations not only occur in the context of tool interoperability but also in early design phases where a team of designers have to negotiate and agree on common component interfaces. Both cooperative work situations have common characteristics and require similar support by a concurrency control component. Since the execution sequence of tool activities is usually not known in advance, predefined design flows are hard to apply. Since the synchronization of design tools mainly depends on the design data itself, support of cooperative work requires a data-oriented approach in contrast to the more control-oriented nature of design flows.

In this paper, we concentrate on the support of cooperative work of design tools in a design environment. We distinguish design tools in their cooperation capability and capture their characteristic access and conflict behaviour taking advantage of an information model. In addition, we provide means to explicitly handle conflict situations to support designers and tools in interactive conflict resolution.

This paper is structured as follows. After relating our work to other approaches, we introduce an information model as conceptual description of a shared object repository in section 3. Section 4 describes our model for cooperative concurrency control. Section 5 discusses some implementation aspects. Section 6 presents our conclusions.

2 Related Work

Several models have been proposed to tackle the problem of concurrency control for CAD applications [4]. Many of them use workspace concepts [5][6] or hierarchies of transactions [7][8]. Transactions are associated with private workspaces which are usually managed using check-out/check-in mechanisms in combination with non-

volatile locks in a public database. In case of access conflicts models using workspaces often react with the creation of new versions of data. This policy transfers the problem of access conflict resolution to the problem of merging different versions. Cooperative transaction hierarchies [7] arrange transactions in a hierarchy. Cooperative work is supported by providing protocols to sibling transactions as well as parent transactions. These protocols allow to relax isolation and serializability of classical transactions. In addition, cooperative transaction hierarchies specify allowed as well as conflicting operation sequences by finite-state automata. Unfortunately, operation sequences of cooperating design tools are often not known in advance or very awkward to specify.

While existing approaches concentrate on methodology management related aspects of cooperation, we aim at support of tight cooperation of design tools. Therefore, we do not primarily look at the hierarchical decomposition of a design project into subtasks. Instead, we consider a set of concurrently running design tools cooperating on a shared design data repository.

3 Information Model

We use an information model to describe data shared by design tools. Note that an information model makes no assumption about a specific implementation. Data may be stored in a design database, in design files, or elsewhere. This allows to describe data on a conceptual level abstracting from specific implementations.

An information model is described in terms of modelling constructs provided by an underlying data model. A data model has to fulfill a minimum of requirements to be suitable for our model. It has to support object types and relationships. Object types may be arranged in an inheritance graph. Each object type *OT* has a set of attributes and a set of relationships. Each relationship *REL* is a tuple of a relationship identifier and a reference to an object type. These requirements are fulfilled by existing object-oriented data models (e.g. [9]) and are also provided by the information modelling language EXPRESS [10]. Both are used for modelling of design data in design environments.

We use a simplified but typical ECAD structure as an example. Figure 1 shows part of an information model describing a simplified design representation for high-level synthesis. In the left part, the object types of a dataflow graph are shown. They represent a simple behavioural description of a design. The object type representing a dataflow graph (*DFG*) has a relationship (*has_op*) to a set of operator nodes (*OPERATOR*) which in turn may refer (*of_dfg*) to other dataflow graphs. In the right part, a hierarchical netlist is depicted. It represents the structural description of a register-transfer level design. A cell (*CELL*) contains (*has_inst*) instances (*INSTANCE*) which are derived from (*of_cell*) other cells. A cell without instances is a library cell. The object repository is described by such an information model.

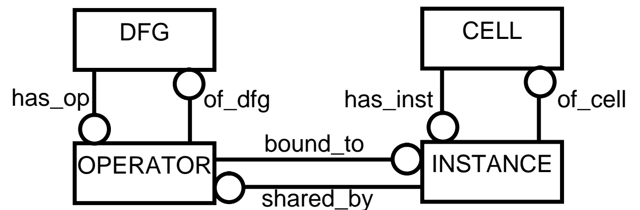

Fig. 1: Example for a simplified information model

A high-level synthesis tool has the task to produce a register-transfer level netlist from a behavioural description of a design. One subtask of the synthesis tool is to bind operators (e.g. add) to functional units on register-transfer level (e.g. an instance of an adder cell). More than one operator may be mapped to a specific functional unit. Therefore, we need a relationship from operators to instances (*bound_to)* and a corresponding inverse relationship (*shared_by*). Such non-isomorphic mappings are characteristic for many ECAD structures.

4 A Model for Cooperative Concurrency Control

Each design tool typically performs different kinds of activities. An *activity* is a sequence of operations performed by a tool on data. For example, an activity of a simulator may be ‘read a netlist’ or ‘write a simulation result’.

Cooperative work of design tools is complicated by the fact that tools have different capabilities to cooperate and to react on conflicts. For example, tightly integrated tools differ from encapsulated tools, interactive tools from batch tools.

Interactive tools (e.g. editors) allow more flexible conflict reactions, since designers may be directly involved in conflict resolution processes. The schedule of activities is determined interactively by the designer in dependence of the state of design data. Therefore, interactive tools primarily have designer controlled schedules.

A batch tool (e.g. a simulator) does not allow for direct designer interaction. Therefore, the schedule of activities is determined by the algorithm of the tool. It may be independent of or dependent on the state of design data. In the latter case the run-time schedule is not known in advance.

Both types of tools differ in their ability to handle conflicts. An interactive tool allows a designer to directly react on conflicts. This requires to make a designer aware of conflicts by appropriate means. Although batch tools do not allow direct designer interaction, they may also react on conflicts if the algorithm of the tool is prepared to do it.

In the following, we consider a cooperative work scenario consisting of a set of concurrently running tools sharing common data. Some tools are controlled interactively by designers, some are batch tools. In our example, we use design tools having the following tasks:

- An editor is used to enter and edit a dataflow graph. The tool is interactively controlled by a designer.
- A synthesis tool is a batch tool and creates a netlist from a dataflow graph. It establishes non-isomorphic bindings between operators and instances.
- A simulation tool analyses the synthesized netlist. Simulation results have to be back-annotated to the dataflow graph in order to allow the designer to evaluate the synthesis result and to change the dataflow graph using the editor.

4.1 Conflict Reactions

We use locks as a basis for the concurrency control component. At least read (R) and write (W) lock modes are provided. Compatibility of locks is determined by a compatibility matrix as known from databases. An *access conflict* occurs if an activity tries to obtain a lock on data that is locked by other activities in incompatible lock mode. We call activities causing conflicts *conflicting activities*.

In our model, an access conflict produces a conflict reaction. We distinguish two types of conflict reactions - isolated and cooperative ones. *Isolated conflict reactions* do not need any feed-back from conflicting activities. The following isolated conflict reactions are provided:

- *abort*. Abort the current activity.
- *wait*. Wait until the required lock is released by all conflicting activities (pessimistic protocol).
- *ignore*. Ignore the detected conflict and continue with unsafe data (optimistic protocol).

Isolated conflict reactions have in common that they do not try to communicate with conflicting activities. As a consequence, the corresponding tools resp. designers are not aware that their activities are producing conflicts.

Since our goal is to support cooperation and conflict resolution, we provide an additional type of conflict reactions. In contrast to isolated conflict reactions *cooperative conflict reactions* allow to communicate with concurrent activities. We provide the following cooperative conflict reactions:

- *request*. The required lock is requested from all conflicting activities. The activity may continue after issuing a request.
- *notify*. A notification is sent to concurrent activities. No lock on data is obtained in this case. The accessed data itself is used as communication channel between activities.
- *kill*. An abort is sent to conflicting activities. This reaction should be avoided in cooperative environments.

The capability of a tool to handle cooperative conflict reactions depends on its type. Interactive tools as well as batch tools with conflict handling algorithms are able to react on conflicts. Cooperative conflict reactions allow to make designers or tools aware of occurred conflicts. This is an essential prerequisite for successful conflict resolution. Designers are often able to evaluate and resolve occurred conflict situations immediately. For example, a designer controlling an interactive tool may rearrange the schedule of activities on the fly.

4.2 Activity Types

Since each tool performs a dedicated task, the activities of a tool usually have a specific and often known behaviour in relation to data. We distinguish *access behaviour* and *conflict behaviour*. Many design tools typically access data by navigating along relationships within the object repository. We exploit this fact and describe characteristic accesses of tool activities in terms of the information model. The access behaviour specifies which inter-related set of objects are accessed by an activity. Furthermore, relationships are used as paths to propagate notifications in the object repository. Therefore, they are important for inter-tool communication, e.g. back annotation or change notification.
In addition to access behaviour, conflict behaviour determines which conflict reactions are performed if conflicts occur during access to inter-related data. We define an *activity type AT* to specify the behaviour of tool activities:

AT = (LM, SOT, SCR, AP).

LM is the required lock mode.
SOT is the start object type the activity type belongs to.
SCR is the conflict reaction executed if a conflict on the start object occurs.
AP = { (REL, CR) } is an access pattern. Each element of AP is a tuple of a relationship REL of the information model and a conflict reaction CR. The

conflict reaction is performed if a conflict on an object REL refers to occurs.

An access pattern specifies a path between inter-related object types. During run-time this path is used to preclaim locks. If an activity executes an operation on an object in a specific lock mode, the activity type belonging to this *start object* is selected. The access pattern of the activity type determines which relationships are used to propagate the lock beginning from the start object. It allows to obtain locks on an inter-related set of objects within a single operation.

The second purpose of activity types is to specify conflict behaviour. We distinguish between a direct conflict on the start object and conflicts occurring during preclaiming. The start conflict reaction *(SCR)* is performed if a conflict occurs on access to the start object. If conflicts occur during preclaiming, the conflict reactions specified by the access pattern are automatically executed. Therefore, one result of preclaiming is information about all occurred conflicts. The designer or the tool may evaluate this information and then decide whether to start conflict resolution or to abort the activity.

Activity types have further advantages. Their comparison allows to identify activities that are free of conflicts as well as activities that may produce conflicts. This information can be used by the concurrency control component to select a pessimistic or an optimistic concurrency protocol. Furthermore, activity types are arranged in an inheritance tree. This allows to define generalized activity types which may be specialized. The resulting classification of typical activity behaviour is a very suitable conceptual basis to precisely characterize design tools in a design environment.

We use our information model in conjunction with the set of cooperating design tools to illustrate the application of activity types. We assume that the editor reads and writes a dataflow graph. Therefore, it needs access to a dataflow graph and the operators it contains. The editor requests locks if conflicts occur. The activity types describing this behaviour are:

$AT_{\text{Editor_read}}$ = *(R, DFG, request, (has_op, request))*
$AT_{\text{Editor_write}}$ =*(W, DFG, request, (has_op, request))*

These activity types specify that on access to a dataflow graph all operators connected by *has_op* relationships are implicitly accessed. If a conflict occurs on the start object of type *DFG* or on objects of type *OPERATOR*, the required lock is requested from conflicting activities.

As second example for activity types, we take the synthesis tool which needs read access to the whole data-

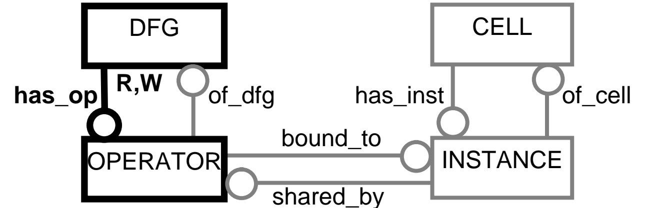

Fig. 2: Activity types of a dataflow graph editor

flow graph and write access to the hierarchical netlist. The tool operates in batch mode and is not able to handle conflicts interactively. Therefore, we select *abort* as conflict reaction. The corresponding activity types are:

$AT_{\text{Syn_read}}$ = *(R, DFG, abort, (has_op, abort))*
$AT_{\text{Syn_write}}$ = *(W, CELL, abort, (has_inst, abort), (of_cell, abort))*

During synthesis the relationships *bound_to* and *shared_by* are established. These accesses need not to be specified by additional activity types. Activity types are primarily used to specify accesses to sets of inter-related objects.

The batch simulator needs a whole hierarchy of cells with all their instances to perform a simulation. If conflicts occur, the simulator waits for locks before it starts:

$AT_{\text{Sim_read}}$ = *(R, CELL, wait, (has_inst, wait), (of_cell, wait))*

In our example, simulation results are written to instances. For design optimization a feed-back from the synthesized netlist to the dataflow graph is essential. A possible feed-back useful for design evaluation is a back-annotation of simulation results to all operators bound to a specific instance. Activity types offer flexible and simple means to describe such tool interactions. We only have to specify a notification along the *shared_by* relationship if a

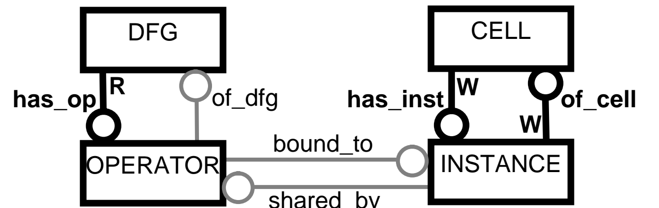

Fig. 3: Activity types of a simple synthesis tool

simulation result is written to an instance. The corresponding activity type is:

$$AT_{\text{Sim_write}} = \textit{(W, INST, abort, (shared_by, notify))}$$

The notification is sent to the editor running concurrently to the simulator. It may be made visible in the editor, for example by high-lighting the corresponding operators.

As shown in this application example, activity types specify characteristic access and conflict behaviour of tool activities. The exploitation of the information model allows to look at access synchronization and inter-tool communication from a common data-centered point of view. This is an advantage in cooperative work scenarios with no given control-oriented design flow.

4.3 Access States and Primitives

Since we aim at support of more conflict-free cooperation between concurrent tools, we have to handle conflicts explicitly in order to resolve them. Our approach is to supplement locks with explicit information about conflicts. We introduce *access states* to capture this information. Each activity has exactly one access state for each object. The access state is either a *locked state* or a *conflict state*.

Primitives are abstractions of data manipulation operations with identical effects to access states. For example, a *lock* primitive in read mode abstracts from all read operations. A *lock* primitive in write mode represents all update and delete operations. In addition to *lock* and *unlock* primitives, we define primitives that allow to explicitly handle lock conflicts. These primitives are used to define communication protocols between concurrent activities involved in a conflict.

- The *request* primitive has already been mentioned as cooperative conflict reaction. It starts a conflict resolution process after conflict detection. The ultimate goal is to obtain the required lock during communication with conflicting activities. A request may be cancelled using the *unrequest* primitive.
- The *grant* primitive allows to pass a lock temporarily from one activity to another that issued a corresponding *request*.
- *Notify* is used to distribute notifications to concurrent activities. A notification does not cause changes in access states. It is used for data-centered communication between concurrent activities. Relationships may be used as paths to propagate notifications. This general mechanism is used for change notification, back-annotation, and general purpose inter-tool communication.
- *register* and *unregister* are used to get access to objects without obtaining locks, e.g. for browsing purposes.

Figure 4 gives a simplified overview of the access states and the primitives supported by our model. Each box represents an access state. A box drawn inside another is a specialized access state. Therefore, the enclosing rectangle represents the most general access state (*arbitrary state)*. A *locked state* is one of the states *locked*, *requested,* or *granted*. If a primitive is defined for an access state, it is also valid for all its specialized access states.

Primitives cause access state transitions. We distinguish explicit and implicit access state transitions. *Explicit access state transitions* are caused by execution of a primitive on an object by an activity. They are depicted as bold arrows. Depending on the primitive, the access state of the object is changed, e.g. the *lock* primitive causes one of two explicit access state transitions. If no conflict occurred, the resulting state is a locked state, either *locked* or *requested*. The latter is a locked state indicating the existence of a request. If the lock primitive detects a conflict, the resulting access state is *conflict*. As possible reaction to this conflict, the *request* primitive may be issued. In this case, the access state transitions to request and the access states of all conflicting activities are implicitly changed from *locked* to *requested.* We call such transitions caused by concurrent activities *implicit access state transition*. They are depicted by thin arrows. The condition for the implicit access state transition labels the corresponding arrow. These transitions can be made visible to the activities by appropriate means. Interactive tools may pop-up conflict messages, batch tools with conflict resolution algorithms may get exceptions.

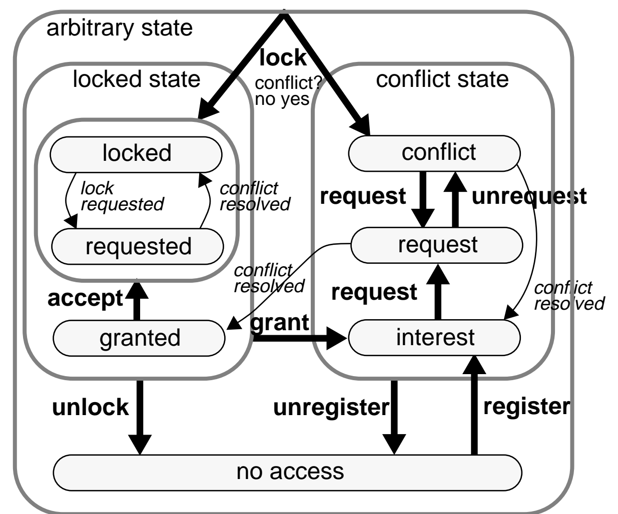

Fig. 4: Access state transition diagram

5 Prototype Implementation

We have implemented a prototype of our model for cooperative concurrency control in C++ and the extension language Tcl [11]. The extension language provides interactive access to the concurrency control component. Integrated design tools access the component using a C++ library which provides an interface supporting the described primitives. Since many design tools do not allow for modification of their tool code, they have no direct access to this interface. These tools have to be encapsulated by wrappers that allow to perform cooperative conflict reactions.

We tested our concurrency control component in a sample scenario consisting of several design tools. A simplified example of the scenario and the underlying information model was used as example in this paper. The concurrency control component is used by a proprietary synthesis tool [12] and a multi-level simulator [13]. These tools have direct access to the concurrency control component and cooperate on a partly shared design. Their typical access and conflict behaviour is captured by activity types. In addition, these tools cooperate with a commercial design compiler which was encapsulated via a wrapper. This wrapper allows to perform conflict resolution on behalf of the encapsulated tool.

The mechanisms provided by our model are used to control concurrent access of these design tools to shared inter-related design data. In addition, communication between the integrated simulation tool and an interactive editor is done via the concurrency control component. Our integrated approach to both aspects allows for easy and targeted tool cooperation.

6 Conclusions

We presented a model for cooperative concurrency control that supports cooperative work of concurrent design tools in a design environment. Our approach combines access synchronization and inter-tool communication in an integrated model. In addition, we introduced activity types which utilize an information model to specify characteristic access as well as cooperation behaviour of design tools. Activity types allows to guide the concurrency control component in access synchronization, conflict avoidance, early conflict detection, and flexible conflict reactions. We support conflict resolution by providing communication primitives. They are used to define protocols which allow designers or tools to communicate directly as conflicts occur, resulting in a more conflict-free tool cooperation.

We have tested our model in a prototype scenario of integrated as well as encapsulated design tools. The concurrency control component is used to organize concurrent accesses as well as inter-tool communication in this scenario. The data-centered point of view taken in our model simplifies the support of cooperative work in the absence of a previously known design flow.

Our model does not aim at fully automatic conflict resolution. Our intention is to provide designers and tools with as much information about conflicts as possible to improve and speed up their decisions on conflicts. Future work will investigate possibilities to support automatic conflict resolution processes to further improve conflict-free cooperative work in design environments.

References

[1] P. van der Wolf, “Electronic CAD Frameworks”, *Kluwer Academic Publishers*, Boston, 1994

[2] T. J. Barnes, D. Harrison, A.R. Newton, R.L. Spickelmier, “Electronic CAD Frameworks”, *Kluwer Academic Publishers*, Boston, 1992

[3] S. Kleinfeldt, M. Guiney, J. Miller, M. Barnes, “Design Methodology Management”, *Proc. of the IEEE*, Vol. 82, No. 2, February 1994, pp. 231-250

[4] N.S. Barghouti, G.E. Kaiser, “Concurrency Control in Advanced Database Applications”, *ACM Computing Surveys*, Vol. 23, No. 3, March 1991, pp. 1-60

[5] R. Lorie, W. Plouffe, “Complex Objects and their Use in Design Transactions”, *Proc. Databases for Engineering Applications, ACM Database Week*, May 1983, pp. 115-121

[6] F. Bancilhon, W. Kim, H.F. Korth, “A Model of CAD Transactions”, *Proc. of the VLDB Conference*, 1985, pp. 25-33

[7] M. H. Nodine, S. B. Zdonik, “Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications”, *Proc. of the VLDB Conference*, 1990, pp. 83-94

[8] I. Widya, T.G.R. van Leuken, P. van der Wolf, “Concurrency Control in a VLSI Design Database”, *Proc. of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 357-362

[9] M.E.S. Loomis, “The ODMG object model”, *Journal on Object-Oriented Programming*, June 1993, pp. 64-69

[10] ISO TC184/SC4 Document N151. “The EXPRESS language reference manual”, September 1992.

[11] J. K. Ousterhout, “Tcl and the Tk Toolkit”, *Addison-Wesley Publishing Company,* Reading, MA, 1994

[12] A. Bredenfeld, R. Camposano, “Tool Integration and Construction Using Generated Graph-Based Design Representations”, *Proc. of the 32nd ACM/IEEE Design Automation Conference*, 1995

[13] W. Meyer, R. Camposano, “Fast hierarchical multi-level fault simulation of sequential circuits with switch-level accuracy”, *Proc. of 30th Design Automation Conference*, 1993, pp. 515-519