ON DETECTING DEADLOCKS IN LARGE UML MODELS

Based on an Expressive Subset

Michael Kersten and Wolfgang Nebel
University of Oldenburg
Department of Computing Science
michael.kersten@informatik.uni-oldenburg.de, nebel@informatik.uni-oldenburg.de

Abstract: The paper describes a method for the detection of deadlocks in large UML models of reactive systems. Therefore a multi-phase-approach will be presented which consists of the four phases: property extraction, potential deadlock analysis, deadlock reachability analysis and result visualisation.

Keywords: Deadlock detection, cycle detection, graph theory, reachability analysis, UML.

1. INTRODUCTION

In the last few years the Unified Modeling Language (UML) emerged to the standard modelling language in the field of object-oriented design. Even in technical domains like automotive and aircraft industry the application of UML has grown appreciable. Due to the fact of high safety requirements in these areas the combination of UML and formal methods is a popular object of research.

A very specific but practically relevant part of this field of investigation is the detection of deadlocks in UML models. Since they are very easy to model but hard to find by manual inspection, automatic methods for deadlock detection are convenient to reduce development costs and to enhance quality. Recent contributions (e.g. [3],[8]) allow the detection of deadlocks in UML models using model checking techniques. They are very general and allow the proof of absence of deadlocks as well as many other properties. Currently the systems to be checked are limited in size (see [1],[2],[6]) depending on the state representation used. A further

restriction of the regarded contributions is the small supported subset of UML.

Hence, the model checking approach cannot be used for any given project without additional effort. In particular the rich set of expressive modelling concepts provided is the payoff of UML.

The presented contribution aims at the automatic detection of deadlocks in large reactive system designs modelled by a rich UML subset.

Therefore in section 1 an expressive UML subset for reactive systems, containing class, statechart, sequence and collaboration diagrams is introduced.

Based on this, in section 2, a multi-phase analysis method is provided, which allows the automatic detection of deadlocks. Subsequently we conclude the presented work.

2. AN EXPRESSIVE UML SUBSET FOR REACTIVE SYSTEMS

In order to provide an expressive UML subset for reactive systems with the full usability it is necessary to include the most commonly used concepts. A minimal but complete subset of necessary concepts is hard to define by academic means. In the present case the concepts included are the achievement of an academia/industry co-project described in [5]. These are essentially concepts for the structural and logical decomposition, modelling of concurrency/parallelism aspects, and the definition of the behaviour of the system under development.

For the definition of the static structure the class diagram concepts of UML are used. Within class diagrams the package concept is used to provide a high-level logical decomposition of the system under development. In order to define the system border the package level stereotype $\ll system \gg$ is used. The further decomposition of the system under development is done using classes and associations.

One important aspect of reactive systems is the concurrency. The first design decision in the reactive system design is the distinction between active and passive components. Therefore the UML provides the feature *isActive* which is anchored as an attribute of *class* in the UML metamodel. Following the definition of the UML specification (see [7]) a class is an active class, if its meta-attribute *isActive* is set to 'true'. The class thus has an own thread of control. Whether active classes on the same hierarchy level of a UML model are executed concurrently or in parallel is left open on this level of abstraction. In the later design phases this aspect may be refined using the stereotypes $\ll task \gg$ and $\ll hw \gg$. The former is used to denote software components which

are running concurrently and the latter identifies hardware components which are running in parallel by default.

For the hardware/software-codesign more elaborate concepts are necessary which are beyond the scope of the present contribution.

Besides the structural modelling of the system under development the behavioural aspects are important. In our profile the internal behaviour of objects (instances of classes) is modelled using statemachines. For each active class a statemachine must be defined. Since passive classes have no own thread of control they have no own internal behaviour and therefore they have no associated statemachine. This does not apply for their operations. The behavioural modelling of class operations may be done using statemachines. For simplicity we state the following assumtions for the UML models under analysis:

1 Nested statecharts are not used,

2 Concurrent states are not used,

3 All active objects are created in the initial phase, later on, only passive objects are created using constructor methods,

4 Guards are boolean expressions over attribute and event-parameter values,

5 Time events are regarded as equal to completion events,

6 No usage of history states.

The assumptions 1-4 are only stated for simplicity reasons and do not limit the generality. Assumption 5 has no influence on the generality, too, but it is needed for the analysis. If time events are not regarded[1] as completion events the underlying deadlock model must be changed into a timed deadlock model. This would introduce a lot of effort without advancing the analysis.

Assumption 6 deliberately increases the generality of the modelling because from the authors standpoint the usage of the history state concept of the UML decreases the comprehensibility of the respective model in an irresponsible manner. Another important aspect of the reactive system modelling is the time modelling. In particular for real-time systems this is essential. For the purpose of deadlock detection we abstract from the timing aspects and therefore do not present our UML time modelling concepts (see [4]) within this paper.

[1] Since in the deadlock analysis time events are only regarded as completion events, the developer may use them as accustomed.

3. A MULTI-PHASE AUTOMATIC DEADLOCK DETECTION METHOD

Our contribution is based on a multi-phase analysis process as shown in Fig. 1. In the first phase the deadlock relevant properties of the UML

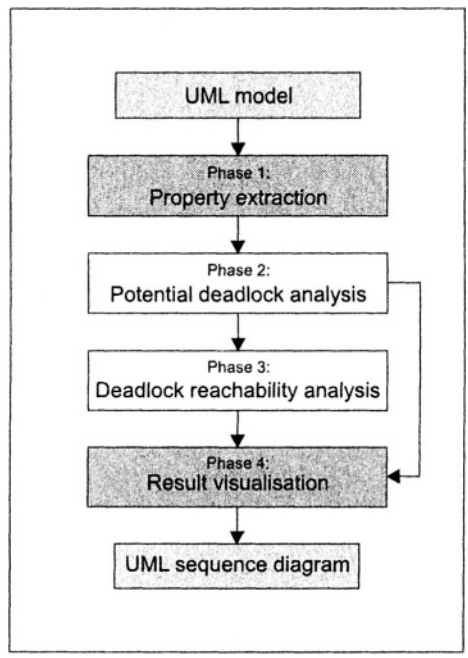

Figure 1. The multi-phase-method

model are extracted and stored in mathematical structures (e.g. lists, sets, tuples) which allow effective algorithms in the second phase.

There the UML statechart diagrams are analysed statically in order to find out which "wait-for"-relations between active objects exist. For this purpose the State-Wait-Graph is introduced which allows white box deadlock detection. In contrast to classical wait graphs, the State-Wait-Graph representation considers the internal state of the active objects expressed as statechart diagrams. The detection of cycles in the State-Wait-Graph indicates the existence and location of potential deadlocks.

Whether these potential deadlocks are reachable at run-time, depends on the binding of attributes and parameters of operation calls between active objects to actual values. These aspects are analysed in the third phase of the method, which performs a deadlock reachability analysis. For this purpose relevant execution paths are derived from the model and the associated transitions are analysed.

When a potential deadlock situation is reachable, this is detected by the method and the complete history of this deadlock can be stated. The included result visualisation mechanism generates a sequence diagram containing an illustration of the deadlock trace.

3.1 Property Extraction

In the property extraction phase all active classes of the UML model are analysed concerning their communication aspects. In concrete terms this means that for each active class the set of produced events and consumed events is calculated and stored in producer and consumer lists. For this purpose call events are regarded as consumer and call actions are regarded as producer of events. Thereby events are only considered if their consumer and producer are suitably associated.

3.2 Potential Deadlock Analysis

A potential deadlock is a cyclic wait situation between concurrent or parallel components of a system each within a specific state. Our respective deadlock model is the State-Wait-Graph (see Fig. 2). This is a directed graph in which each vertex represents an active object in a specific state (of the associated statechart diagram) and each edge represents a 'wait-for-relation'. The number of vertices of the State-

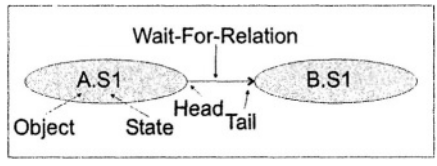

Figure 2. Principle of a State-Wait-Graph

Wait-Graph is the same as the number of states of all statechart diagrams of the model which are associated to active classes. The number of edges depends on the transitions defined in these statecharts. The head of each edge is connected with the vertex waiting for some specific event. The vertex connected with the tail of the edge is a potential producer of this particular event.

The potential deadlock analysis starts with the creation of a State-Wait-Graph. This is done by operating on the structures (e.g. consumer and producer lists) created in the property extraction phase.

Having created the full State-Wait-Graph of the system, potential deadlock situations can be detected. A potential deadlock situation is

a situation in which two or more objects (each in a specific state) are waiting mutually for the production of a particular event.

In our approach the next phase is to detect all cycles in the State-Wait-Graph and then sort out the relevant ones. In contrast to classical wait-graphs (e.g. [9]), where each detected cycle is a potential deadlock the procedure is more complicated for State-Wait-Graphs due to their white-box-nature. Cycles with all vertices being of the same object (as shown in Fig 3) are not potential deadlocks but logical errors in the corresponding statechart diagram. Cycles with two or more vertices of different objects

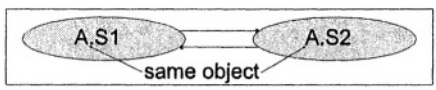

Figure 3. Cycle within one object

are potential deadlock situations, if no object is involved with more than one state. Otherwise we denote them as 'cycles with over-involved objects' (see Fig. 4). The detection of all cycles in the State-Wait-Graph

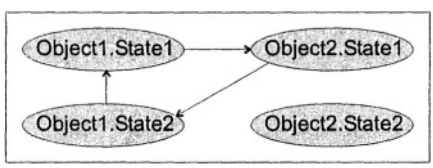

Figure 4. Over-involved object

is done using an advanced depth-first-search algorithm which calculates all cycles of the graph and the partitioning within a single run. The pruning of logical errors and cycles with over-involved objects is done using set-operations.

After the pruning the remaining potential deadlock situations need to be examined concerning outgoing edges. This is done by traversing the graph and calculating the out-degree of each vertex. If all vertices have the $out - degree = 1$ the potential deadlock situation is called a potential deadlock in our approach. If there are vertices with an $out - degree > 1$ (as shown in Fig. 5) it depends on the target object of the edge leading out of the cycle, whether the cycle under examination is a potential deadlock. When the target object is not involved in the cycle the examined cycle is not a potential deadlock. Otherwise it must be checked, if the edge connects two states of the same object. If the check evaluates to true the potential deadlock is combined with a logical error and the logical error should be corrected before going on in the

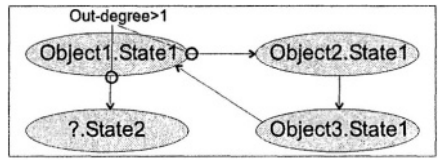

Figure 5. Out transition of a cycle

deadlock detection. If it evaluates to false we have a further cycle in the State-Wait-Graph. In this case the next phase can be initiated because our cycle detection ensures that all cycles are found and all cycles are handled seperately in the further analysis.

Potential deadlock means, that it is statically possible that the deadlock occurs but whether this really may happen at run-time depends on some dynamic aspects of the model which are analysed in the next phase, the deadlock analysis phase.

If no potential deadlock is found in this phase, the system is deadlock free and the next phase may be omitted.

3.3 Deadlock Reachability Analysis

The purpose of the deadlock reachability analysis phase is to provide evidence for each detected potential deadlock found in the previous phase. This means to calculate the possible paths into the potential deadlocks detected using a search algorithm and to analyse whether these paths are executable by an heuristic simulation approach. As illustrated

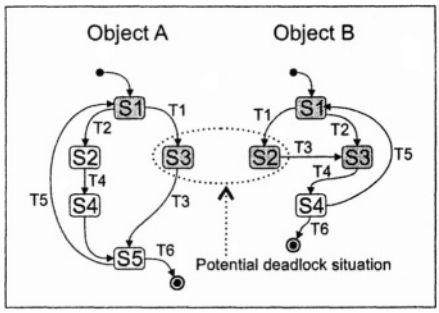

Figure 6. An example of statecharts containing a potential deadlock

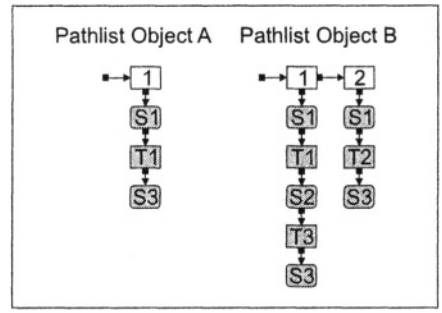

Figure 7. An example of path lists

in Fig. 6, the search algorithm performs a specialised depth-first search for each relevant statechart diagram and stores all paths to potential

deadlock states in a path list (see Fig. 7). Each path list contains all states and transitions of the respective path.

The path lists are used as input for the subsequent heuristic simulation. In the simulation the state machines are executed on a simulator which implements the exact UML semantics as defined in [7]. Besides very special features like the queue length and the evaluation order of expressions as defined in the semantics the following dynamic aspects of the model need to be considered: guard conditions, event parameter values and attribute values.

In the case that one or more paths into a potential deadlock situation exist it is sufficient to find one executable path during the simulation. The other case is the more expensive one. When all paths of the path list are executed once and no executable path is found, it cannot be stated whether there is an executable path or not. Even after theoretically infinitely many executions one cannot be sure that the next execution does not lead to a potential deadlock situation. In this case the simulation will break after an adjustable number of n executions and a corresponding warning is issued to the user.

The disadvantage of this heuristic simulation approach is that the nonexistance of deadlocks cannot be proved if there are potential deadlocks found in phase 2 into which no executable path could be found after n executions.

In this infrequent case only an exhaustive search over a completely unfolded state space as model checking does could help. Since our approach basically adresses models which are to large for model checking there is no other solution than the usage of manual abstraction techniques in conjunction with advanced model checking techniques.

3.4 Result Visualisation

The last phase of the presented contribution is the result visualisation phase. Its task is to provide an easily comprehensible representation of the results of the deadlock detection procedure. Therefore extended UML sequence diagrams are applied. In Fig. 8 a simple deadlock situation is shown, in which, for simplicity, only the deadlocked states 'A.S1' and 'B.S2' are displayed. The corresponding result visualisation chart is

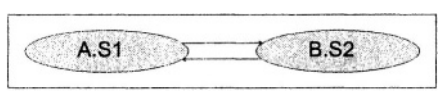

Figure 8. Cycle in the State-Wait-Graph

given in Fig. 9. There it is apparent that the active objects A and B are

running directly into a deadlock situation after being created from the system. Next to the sequence of communications ahead of the deadlock

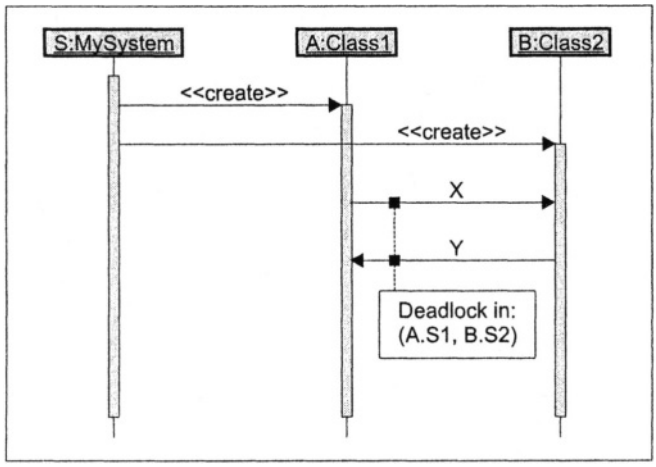

Figure 9. Example of a deadlock trace

situation the participating objects and states are relevant and presented to the user. Other visualisations are possible but not in the focus of this contribution.

4. CONCLUSION

We have presented a method for the detection of deadlocks in UML models of reactive systems. The present approach has two main advantages. First, the method supports an expressive set of UML concepts and thus real-world UML models of reactive systems can be checked without modification. Second, the size of models to be checked, may be significantly larger than a model checker can handle. Since the evaluation of the presented method is in progress, we will not give a full complexity analysis within the scope of the presented contribution but a rough estimate causes the assumption that the overall algorithm will be linear in the number of states and transitions multiplied by a factor depending on the number of potential deadlocks found, the number of reachable execution paths and a set of further internal parameters.

The main drawback is that in seldom cases (as discussed in section 3.3) the freedom of deadlocks cannot be stated and thus the method is not complete. Since in this case a corresponding message is issued to the user the creditability is not affected.

Another disadvantage is the loss of generality in contrast to model checking, which allows the proof of almost any property expressible in the particular logic dialect used.

For a class of applications the criteria model size and expressiveness of the supported UML subset are more important than the generality of properties.

The property extraction and potential deadlock detection phases are experimentally evaluated using a prototype implementation. The first results of this evaluation are promising, but it turned out that the creation of adequate UML models as input for the method is an extensive and difficult task. Hence, we defer the continuation of the experimental evaluation in favour of a formal proof.

REFERENCES

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Information and Computation,* volume (98)2, pages 142–170. IEEE, 1992. also in 5th IEEE LICS 90.

[2] E. M. Clarke and H. Schlingloff. Model checking. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning,* chapter 21, pages 1367 – 1522. Elsevier Science Publishers B.V., 2000.

[3] Alexandre David, M. Oliver Möller, and Wang Yi. Verification of UML Statecharts with Real-time Extensions. Technical report, Department of Information Technology, Uppsala University, http://www.it.uu.se/research, February 2003.

[4] Michael Kersten, Ramon Biniasch, Wolfgang Nebel, and Frank Oppenheimer. Die Erweiterung der UML um Zeitannotationen zur Analyse des Zeitverhaltens reaktiver Systeme. In R. Drechsler et al., editors, *GI/ITG/GMM Workshop 2003,* pages 11–20, Bremen, February 2003. Shaker Verlag.

[5] Michael Kersten, Jörg Matthes, Christian Fouda, Stephan Zipser, and Hubert B. Keller. Customizing UML for the Development of Distributed Reative Systems and Ada 95 Code Generation. *Ada User Journal,* 23(3), 2002.

[6] K. L. McMillan. *Symbolic model Checking.* Kluwer Academic Publishers, 1993.

[7] OMG. OMG Unified Modeling Language Specification (version 1.4). Technical report, OMG, http://www.omg.org, September 2001.

[8] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science,* 47, 2001.

[9] Mathias Weske. *Deadlocks in Computersystemen.* International Thomson Publishing GmbH, Bonn, Albany [u.a.], first edition, 1995. ISBN 3-929821-11-7.