Abstract

System-level design requires behavioral models of its complex components in order to validate the designs, synthesize implementations and generate tests. Descriptions of these components are often only available as English descriptions. Developing behavioral models from natural language descriptions has been a tedious and time consuming task. This paper describes the semiautomatic generation of conceptual models from text and a simulator for these models. The conceptual modeling language is a knowledge representation notation (semantic hypergraphs), and simulation of these abstract descriptions based on the VHDL execution model is described. Simulation at the conceptual level avoids having to add the extraneous declarations and other syntactic artifacts necessary with more conventional description languages. These declarations not only take time, but also prejudice the implementation.

1. Introduction

To use or reuse components in new systems designs requires they have behavioral models to support validation by simulation or rule checking, behavioral synthesis and test generation. Unfortunately, formal behavioral simulation models are often unavailable, but instead only English descriptions, supplemented with a block diagram and timing diagrams, may be obtained by the prospective user. This is generally the case for complex commercial devices and unimplemented devices such as those described by proposals and patents. Generating simulation models from such descriptions is a difficult task requiring considerable expertise, but one which does not tend to attract experienced engineers.

The goal of the research described here has been to automate the process of generating behavioral models from natural language descriptions of devices. We call this process **model extraction** after a similar process in message understanding called information extraction [2]. Model extraction involves a number of tasks, some of which have been automated. This process consists of two phases: 1) the extraction of a semantic representation from the English text, and 2) the generation of a formal simulation model in a suitable description formalism. The intermediate representation in model extraction here is a semantic hypergraph, where the nodes represent concepts such as actions, events, states, devices and values, and the hyperarcs represent relationships among concepts.

The first phase of model extraction is rooted in the disciplines of natural language understanding and computational linguistics. The study of information extraction which includes filling conceptual templates from text is particularly relevant to automatic model extraction. As described later, our approach to model extraction involves unifying word sense graphs for sentence words into sentence meaning graphs. Systems for automatic message understanding have been able to extract 70-80% of the desired information in documents with 80-90% accuracy [14]. These systems also perform at about 90% on simpler identification tasks. In an earlier DARPA study [1], human analysts performed at 60-80% in some information extraction tasks. The implication of these results is an automatic model extractor may approach human performance and is certainly a goal worth pursuing.

The second phase of model extraction translates the conceptual graph model into a simulatable formal notation such as VHDL, Petri nets, control dataflow graphs or statecharts [7]. While experiments have indicated that this translation can be automated [9,15,5], these target languages have some disadvantages, including forcing the semantics of the original description into a limited notation. Uninterpreted notations such as Petri nets can be simulated, but are not very expressive and fail to capture much of the description semantics. Control dataflow diagrams [11] are more expressive, but require that the computations be expressed in a dataflow model. Unfortunately, English style does not lend itself to
dataflow expressions. For example, it is rare to find dataflow expressions such as, “The results of the addition of the constant 7 to the multiplication product produces a sum which is an operand to ....” The statechart paradigm [7] is a more complete modeling notation, but has the same drawback of forcing the description into a combined state-oriented, dataflow view.

On examining English descriptions of device behavior, the natural modeling style might be called an Agent/Transaction model. Devices (agents) perform actions and in doing so cause other agents to act. These actions are often nonspecific as represented by verbs such as performs and executes. Causation may be implied by ordering of sentences in the narrative, or by transmission of unspecified signals to other agents which respond by performing certain functions when a signal is received. Causation may also be stated explicitly as in “Execution of the program causes the device controller to enter an idle state.” While the Agent/Transaction style can be forced into VHDL or statecharts styles, that process requires undesirable assumptions and, even then, the model will probably not be complete enough to satisfy syntactic analysis. For example, in the sentence above, a causal signal between an ‘execution process’ and an ‘enter analysis. For example, in the sentence above, a causal signal between an ‘execution process’ and an ‘enter process’ must be hypothesized and declared with an assumed format, resulting in an unwarranted design decision.

Since the intermediate semantic hypergraph, is available, the question naturally arises if it can support simulation and analytic analysis before generating an conventional engineering model. The answer is affirmative, and a simulator for this representation is described later. The conceptual model simulator is an event-driven approach based on a VHDL execution model [10] and also draws on the simulation semantics of statecharts [8]. The general simulation strategy is that event concepts can fire and action concepts can execute. Event and action concepts are in turn enabled or inhibited by state concepts. When actions execute, they can change related value concepts, cause execution of related actions, fire related events, and change the states of devices.

2. Representation of semantics

The meaning of a device description is represented here by a form of hierarchical semantic hypergraph called conceptual graphs [13], which consist of a set of labeled nodes, called concepts, and labeled, directed hyperarcs, called conceptual relations. Concept nodes are labeled by concept types, such as device, value, action, event, state and attribute. These types have subtypes defined by a type hierarchy [4]. For example, processor and memory are device subtypes. Calculate and transfer are action subtypes. The hyperarcs are labeled by relation types such as causes, after, agent, operand, part and enables. Conceptual graphs can be joined by unifying concepts of comparable types.

3. Conceptual model extraction

This section describes the process of deriving a conceptual model from a textual description of a device. First, a conceptual graph is derived for each sentence. This step is performed automatically by the ASPIN semantic analyzer software [6], which has recently been improved and ported to C++. The semantic analyzer first parses the sentences using a context-free grammar developed from a study [4] of microprocessor descriptions and enhanced to deal with descriptions in U.S. Patents. Since English is ambiguous, multiple parse trees are generated for many sentences. Each parse tree is then analyzed semantically to determine if it might be meaningful. Semantic analysis proceeds by fetching word sense graphs (small semantic hypergraphs) from the dictionary. Guided by the parse tree, an attempt is made to unify the word sense graphs. A successful unification is taken as the meaning graph of the sentence.

The next step in semantic analysis is to join the sentence graphs into by identifying concepts having the same referents. Software has been developed for such basic coreference detection [12].

4. Conceptual simulation

An event-driven approach was adopted for simulating conceptual graphs. Since the term event is a concept type in this domain, we will use the term incident to describe the activities of the simulator. Only action and event type concepts are active in simulation. Attribute concepts may be affected by actions, and so are passive participants in simulations. Event concepts represent changes in the activity of actions and can only fire. Action concepts can have one of the following status attributes: {inactive, started, suspended, resumed, finished}. A value concept can have an evaluation attribute such as a constant or literal, and a device can be in a mode (have a state attribute). Finally, a state concept can have a (veracity) attribute of being true or false. Often, however, a state is described by a graph, and then, the veracity of the state is determined by whether its describing graph is a subgraph in the current device description graph. In our simulation approach, the only changes that the simulation can cause in a graph are changes in the attributes of concepts.

The simulator uses incident records propagated over conceptual relations to fire events, execute or change the status of actions, and change the evaluation attributes of values, the veracity attributes of states and the mode attributes of devices. An incident record has the format
i(type, ID, change, delay, level). The type identifies the general type of the target concept, identified by its ID. The change parameter indicates how an attribute of the concept is to be affected. The delay parameter specifies how many time units are to lapse before the incident affects the target. The level is necessary because the hypergraphs can be nested. All else being equal, it is necessary to process all incidents at the innermost nesting first.

The effects of most incident types is to merely change an attribute. One exception is an event incident which does not change an attribute but attempts to fire the target event. An event will fire only if all its enabling relations attach to true states and all its inhibiting relations attach to false states. When an event does fire, it may propagate new incidents over attached relations.

Incidents that terminate or interrupt actions merely change the activity status of the target action. Processing action incidents which invoke activities is more complex. First, the action must be enabled. Initiate and resume incidents not only change the status of actions and generate new incidents by virtue of certain types of attached relations, but may also invoke the execution of procedures from a library of associated action procedures, or may cause the simulation of a referent graph that describes (or decomposes) the action. Since the activity of an action may extend over time, the timing of activities must be considered. If the action has no library procedure or defining graph, and if it has no attached arcs that can propagate terminate or interrupt incidents to it, then the action must self-terminate or self-suspend by issuing an appropriate incident to itself. If the action is described by a graph, then the first behavior of that graph is immediately initiated/resumed. Also, that description graph should generate a terminate or interrupt incident to the action. In the case of an associated procedure, we adopt an execution model similar to the VHDL model [10]. Receipt of an initiate or resume incident invokes the associated procedure once. The difference between these two incident types is only whether the procedure executes its initialization routine or not. The procedure may generate a terminate or interrupt incident to the action, or for iteration it may generate self-resumption incidents. In the latter case, it will be up to relations attached to the action to propagate a suspension or termination incident to escape the loop.

Some of the relation types do not propagate incidents, but may be involved by providing paths for fetching operand values or verifying states. When a value incident is received at a value concept (over a result relation), it immediately changes the evaluation concept adjacent to the value. Similar effects occur with conjunctive relations. Temporal relations may be handled it two ways. First, they may be used as checks on the simulation. Alternatively, an action incident can be propagated, treating before as a causal relation similar to a flowline in a flowchart.

5. Simulator implementation

An event-driven simulator that implements most of the requirements described above has been prototyped in Prolog. The simulator processes incidents and generates new incidents in each cycle until there are no incidents pending, or a specified time limit is reached. The simulator uses a number of sets (lists) in performing a simulation cycle. The queue list contains all incidents that have been generated but not yet processed. At the beginning of a simulation cycle the input interface is polled for any user supplied incidents, which are then added to the queue. Then, all incidents to be executed at the current time and at the deepest level of nesting are removed and placed in the current list. If no incidents are pending for the current simulation time, the simulation time is advanced to the time of the least future pending incident, and incidents at that time are removed to current. In processing the current incidents, order is important. Since actions and events are enabled by states, and the veracity of states can depend on values, it is necessary to first process value incidents and then state incidents. Since states can be described in terms of the status of actions, it is necessary to process terminate and interrupt action incidents next. In processing state incidents received at states described by graphs, the new veracity should be checked by evaluating the truth of the describing graph at that time. If a state has a veracity attribute and is described by a graph, they must agree, or the simulation must stop with an exception. Event incidents may then be processed.

Finally the initiate and resume incidents that invoke procedures as well as generate new incidents can be processed. The first step in processing an activation incident is to decide if the target action is enabled.

All incidents generated in a cycle are immediately placed in a next list. At the end of the cycle, the next list is appended to the queue of pending incidents. During this process, a history of the successfully processed incidents and their simulation times must be maintained for temporal relation checking. This could be done on a cycle-by-cycle basis if there are no delays. In this case, ‘when’ temporal relations mean during the same cycle. With delays present, temporal relations must be checked on the execution history since a single time may span multiple cycles. The simulation cycle is summarized below. An earlier version of a general conceptual graph simulator is described in more detail in [3].
Simulation cycle:
1) Get input incidents
2) Get current incidents from queue (deepest level and soonest time)
3) Process incidents
   a) process value incidents
   b) process terminate and interrupt action incidents
   c) process state incidents and check consistency
   d) process event incidents
      i) check enablement
      ii) generate new incidents if enabled
   e) process initiate and resume action incidents
      i) check enablement
      ii) change status of actions
      iii) generate start-related incidents
      iv) invoke procedure or schema
      v) generate self-terminating incident
         if no procedure or schema
4) update queue with next list

6. Conclusions and future plans

From the work completed thus far, it is apparent that simulation models can be derived at least semi-automatically from English descriptions of digital devices, and that event-driven simulation can be performed using an intermediate knowledge representation. Future work includes evaluating the efficacy of automatic extraction of conceptual graphs from text, integrating the coreference detector with the semantic analyzer, porting the simulator to C++ and constructing a suitable GUI for the analyzer/simulator.

7. Acknowledgements

This project has been funded in part by the National Science Foundation, Grant MIP-9707317.

8. References


