ADL: A Graphical Design Language for Real-time Parallel Applications

Maarten R. van Steen[1,2]
Teus Vogel[2]
Armand ten Dam[2]

[1] Erasmus University Rotterdam
Department of Computer Science
POB 1738, 3000 DR, Rotterdam
steen@cs.few.eur.nl

[2] TNO Institute of Applied Physics
POB 155, 2600 AD, Delft
{vansteen,tvogel,tendam}@tpd.tno.nl

Abstract

Designing parallel applications is generally experienced as a tedious and difficult task, especially when hard real-time performance requirements have to be met. This paper discusses on-going work concerning the construction of a Design Entry System which supports the design phase of parallel real-time industrial application development. In particular, in this paper we pay attention to the development and implementation of a graphical Application Design Language. The work is part of the ESPRIT project Hamlet which focuses on industrial application of transputer-based systems for commercially strategic real-time applications.

1 Introduction

Over the last twenty-five years concurrency has become one of the most active areas of research in computer science. Concurrent models have been widely applied in the design of operating systems and databases, and as efficient implementations of high-level concurrent languages became available, software that was originally coded in an assembly language could now be developed using high-level language constructs yielding well-structured, efficient, and portable implementations.

As insight in the behavior of concurrent models grew, focus has gradually shifted from the problem of developing programs that behave in a well-defined manner to that of developing programs that exploit parallelism to improve overall efficiency. This shift of focus has brought us, somewhat surprisingly, to a stage comparable to the first stages of research in concurrency issues. At the moment, parallel applications are generally written in a highly machine-dependent manner and often violate basic rules of well-structured software in order to retain efficiency [7].

And indeed, developing parallel programs is generally experienced as a difficult and tedious task in comparison to the development of sequential programs. This is not too surprising if one considers the additional requirements that are currently demanded from a parallel application developer. In the first place, he or she must concentrate on the specification of an algorithm such that parallelism can be exploited to a maximum extent. This requires *a priori* insight in the parallel aspects of the problem to be solved. More seriously, however, is the fact that deriving an actual implementation requires that the developer also has knowledge concerning the semantics of communication and synchronization mechanisms, as well as knowledge concerning the target parallel computer on which the algorithm is to be executed.

And things become even worse when one considers the development of real-time applications. In these cases, exploiting parallelism seems an obvious choice. Unfortunately, the additional hard performance requirements that are often demanded in the real-time world make the process of exploiting parallelism no less easier. Besides the fact that application developers are often forced to exploit the target machine to its edges, communication itself should be completely subject to timing constraints. This means that if communication within a certain timespan failed for whatever reason, it should be possible to take special measures in a flexible way. Timed communication and its effects on program development is an issue that is obsolete in most scientific parallel applications.

As it turns out, practice indicates that these additional requirements are quite demanding. Considering the fact that hardly any support is available to assist the structured development of parallel real-time applications, it is not too surprising that there is currently still a strong need for advanced monitoring and debugging systems: the results of a development process can often only be measured in the final stage when a first version of the implementation is actually running on the target parallel machine. This is not an approach that can be followed for very long and as such has been recognized by the companies participating in the ESPRIT project *Hamlet*.

Hamlet focuses on exploitation of parallelism for hard real-time applications. In particular, attention is paid to *industrial* embedded applications for transputer-based systems, and which are developed for commercial strategic reasons. It has been recognized by the Hamlet consortium that if the partners are to maintain their strong market position, advanced practical support for parallel application development is necessary. To this aim, attention is currently being paid to software components that assist during the global and detailed design of applications. It is beyond the scope of this paper to discuss the Hamlet project in detail. To that aim, we refer the interested reader to [3, Chapter 8]. Here, we shall concentrate on just one such component: a so-called *Design Entry System*, and in particular its graphical *Application Design Language*.

2 Design by data flow diagrams

In order to support the *design phase* of parallel real-time application development there are roughly two extremes which can be followed: one can choose to devise a complete new method with accompanying techniques, or otherwise simply use existing methods. The first approach not only requires a great deal of research, it can also be expected that at best many years will pass before a new method is accepted in an industrial environment. The second approach has so far been followed by many application developers. In particular, methods based on data flow diagrams such as introduced by Yourdon [13] and specifically extensions thereof to support real-time developments (e.g., Ward [11]) are now often used as common development methods in industry.

But none of these traditional methods is actually suitable for dealing with parallelism, although their inventors often claim otherwise. The problem, as we see it, is that no distinction is made between *concurrency* and *parallelism* and that the two are often mistakenly taken to be the same. Concurrency, as viewed by us here, is a technique that enables a developer to *model* a system in such a way that its structure and dynamics are reflected in a natural way. Parallelism, on the other hand, is considered by us a means to *exploit a target machine* to meet performance requirements. In other words, where concurrency focuses on modeling the real world, parallelism focuses on implementation for a specific environment. The two need not be easy to combine, as is often illustrated by the design and implementation of concurrent object-oriented languages.

The problem that needs to be addressed then is the development of a design method and supporting tools which:

- are familiar to developers of industrial real-time applications,
- are based on methods that have proved to be applicable in an industrial context,
- deal with concurrency *and* parallelism.

Based on these requirements, we have chosen to support parallel real-time application development based on data flow diagrams (DFDs). However, where DFDs are generally used in the analysis phase of a development project, we have adapted DFDs in such a way that they are more suitable for global and detailed design. In particular, emphasis has been put on the support for design of different communication structures, grouping of logical activities into processes, and integration of data and control transformations. This has resulted in a first version of an *Application Design Language*, referred to as ADL/1.

3 ADL: Concepts and notations

In this section we discuss the main concepts of ADL: processes, activities, and how communication is dealt with. In addition, we present features which are still under development but which will be incorporated in a next version of ADL.

3.1 Processes and activities

A key concept of ADL is formed by *activities*, which are used to model logical entities capable of transforming incoming data according to some control schema. Similar to Mascot [9], the interface of each activity is entirely specified by means of a collection of *gates*. Gates specify either incoming or outgoing data and provide the essential means to connect activities to each other in a structured manner.

From the activity's point of view, data can be either communicated to the outside world by means of an *output gate*, or, conversely, data can be received through a so-called *input gate*. To this aim, three different types of communication are supported:

- *blocked communication*, meaning that an activity cannot proceed until data transfer has actually taken place;

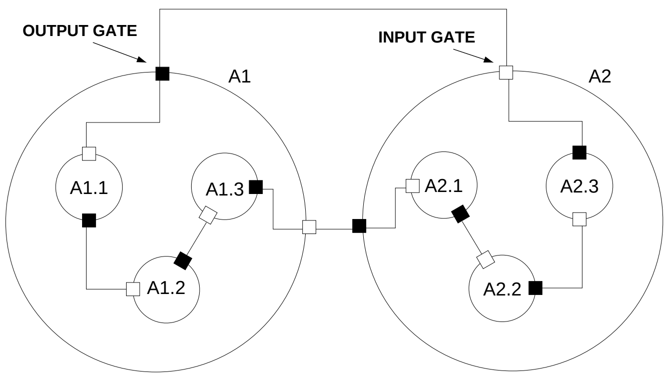

Figure 1: An example of a hierarchically organized collection of activities.

- *non-blocked communication*, meaning that if communication could not succeed immediately the activity will proceed without further delay and without transferring any data;
- *timed communication*, in which case communication should take place within a specified amount of time units.

Blocked and non-blocked communication are in fact special cases of timed communication. If T denotes the specified time an activity is willing to wait before communication can take place, then clearly the case $T = 0$ corresponds to non-blocked communication whereas the case $T = \infty$ is the same as blocked communication. For practical reasons, we have chosen to incorporate all three communication types. Also note that these forms of communication relate to the moment when communication should take place *as required by the communicator*, and if this requirement could not be met communication is cancelled all together. This is different from (a)synchronous communication which involves *all communicating parties*, and which is, in principle, never cancelled. We shall discuss a number of examples in which blocked and synchronous forms of communication are combined in the next subsection.

Activities are represented as circles with input and output gates drawn as white and black boxes, respectively. Also, activities can be hierarchically organized as illustrated in Figure 1 (the connections between gates is explained shortly).

Activities are appropriate for logical design decisions: they represent logical entities which, in principle, can act concurrently. In practice, the number of activities that act concurrently may not correspond to what is desired from an implementation point of view. To that aim, activities can be grouped into *processes*, intended to be the actual units of concurrent behavior in the final implementation. From a conceptual point of view, a process is just another activity, i.e., it models a transformation entity which communicates with other processes by means of gates. The main difference between an activity and a process is that the latter constitutes inherent *sequential* behavior, despite the number of activities it may contain. In other words, processes form the means to *add* sequential behavior in order to fit the logical design in an implementation environment.

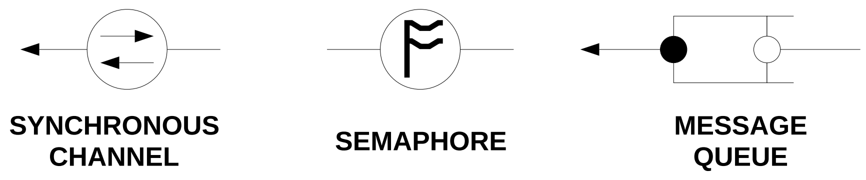

Figure 2: Notations for ADL concepts.

3.2 Communication through protocols

Communication in ADL designs is modeled by so-called *protocols*. Currently, ADL/1 supports three types of protocols: synchronous channels, message queues, and semaphores. Figure 2 shows the notations for the various communication protocols.

Synchronous channels. A synchronous channel is used to model point-to-point communication between two activities and corresponds to the standard synchronous communication means in most message-based programming languages [2]. A synchronous channel is modeled as a directed edge between the output gate of a sending activity, and an input gate of a receiving activity. The important thing to note about synchronous channels is that their functionality is primarily determined by the lack of buffering capabilities. In other words, if two activities communicate data through a synchronous channel, both sender and receiver will have to synchronize. *When* communication may take place is determined by the gates of the respective activities.

For example, imagine a scenario in which a sender wants non-blocked synchronous communication, while the receiver has chosen for blocked (synchronous) communication. In this case, the sender will only transmit data whenever the receiver is capable of accepting that data. On the other hand, the receiver will block until the data is actually transmitted.

Message queues. ADL/1 also provides support for modeling asynchronous communication by means of *message queues*. Message queues in ADL/1 are buffers that act on a first-come first-serve basis and may have either an infinite or finite capacity. Several activities may be connected to a message queue, in particular, an activity that wants to put data into a queue has a link between one of its output gates and the tail of the queue, while a reading activity will have a link between the head of the queue and one of its input gates.

Again, note how the gates determine the conditions under which communication can take place. For example, imagine a message queue Q with finite capacity connected to an output gate of an activity A. Suppose that at time t_0 activity A wants to append data to Q according to a timed communication protocol such that communication should take place before T time units have elapsed. If at time t_0 the queue was full, then this form of communication specifies that if A cannot append its data before $t_0 + T$, it will simply cancel the communication all together.

Semaphores. Finally, ADL/1 also supports semaphores. Obviously, our concept of gates enhances the traditionally semantics of semaphores. For example, a conditional *wait*-operation

[1] is modeled as a combination of a non-blocking input gate and an ordinary *wait*-operation: if the requesting activity cannot retrieve a semaphore token immediately (normally implying that it should wait), it simply continues without further delay. Conditional semaphores are typically used in time-critical applications: a simple trade-off is made between entering a critical region (for which the activity should acquire the token), or otherwise to continue with other tasks.

3.3 An example

Figure 3 shows an example of an ADL design, developed with our current implementation of the language. A total of five activities have been modeled, together with three synchronous channels, two message queues, and a single semaphore. Also, an *environment* has been included, representing components outside the system, but which interact with various activities. Our current implementation only allows to draw syntactically correct ADL/1 designs. For example, it is impossible to draw a synchronous channel connected to three activities. Implementation issues will be further discussed in the next section.

3.4 Enhancements to ADL/1

It should be clear that ADL/1 currently lacks at least two important features: behavioral specification and replication. We shall discuss these two issues briefly here, but note that behavior modeling and replication are still subject to debate within the project. Actual implementation of these concepts has been deliberately deferred to a later stage.

3.4.1 Modeling behavior

Activities form the units of behavior in ADL/1, and obviously there should be a means for supporting the description of behavioral aspects. In most methods based on data flow diagrams, there is a strict distinction between data activities and control activities. Data activities are used for modeling data transformations, whereas control activities describe the system's flow of control. In order to describe data transformations, pseudo-code, or sometimes even a high-level procedural language is used. Control flow is described by means of state-transition diagrams (STDs).

A major drawback of DFDs (or function/data models in general) is that a strict distinction is made between *data objects*, and *functions* that transform that data. Consequently, any change in the data definition may severely affect the definition of functions. Furthermore, when separating control and data transformations, the process of integrating them (which is required for an implementation) may turn to be less straightforward than one would expect. These considerations have led us to integrate data and control transformations into a single activity, making our approach essentially object-based (see also [6]).

Modeling behavior in ADL is done by means of state-transition diagrams. However, where STDs normally consist of a single notion of a state, and transitions between states can only occur as the result of an event, we have chosen to use a form of STDs by which a developer can focus on *communication* entirely. This means that we are not initially interested in data and control transformations which do not immediately relate to parallelism. This perspective has led us to distinguish three types of states. *Communication states* describe the situation in which an activity is involved in communicating data through one of its gates. *Processing*

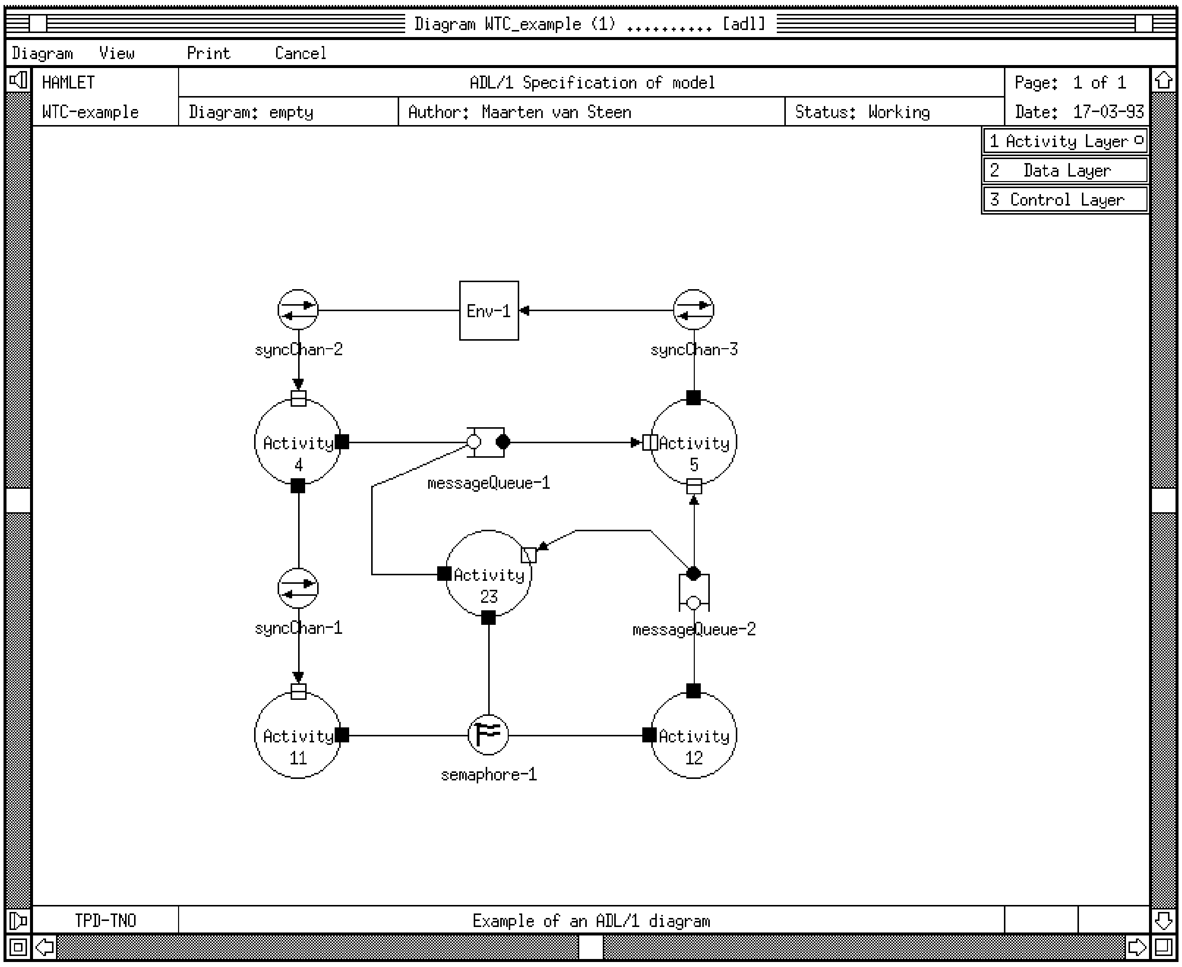

Diagram WTC_example (1) .... [adl]

HAMLET	ADL/1 Specification of model			Page: 1 of 1
WTC-example	Diagram: empty	Author: Maarten van Steen	Status: Working	Date: 17-03-93

TPD-TNO	Example of an ADL/1 diagram		

Figure 3: An example of an ADL design as constructed with the current implementation.

states are used for modeling data transformations exclusively. Finally, *event states* are states in which an activity simply waits until an event occurs.

As mentioned, full development of STDs has been deliberately deferred until a later stage in the project. Consequently, they are not yet supported by our first version of ADL.

3.4.2 Replication

Another issue which is extremely important when dealing with design is the means for indicating that certain activities (or processes) should be replicated. Replication was also introduced by Ward and Mellor [12] as a means to indicate multiplicity of *functionality*. However, when dealing with specifying functionality it is questionable what replication actually means. In the design phase, on the other hand, replication has a clear meaning if we associate each activity explicitly with an *instance*. And this is exactly how activities and processes should be considered in ADL/1. Replication is thus a means for exploiting parallelism. The underlying thought, of course, is that replicated activities indeed return as replicated instances in the final implementation.

Replication is a subject that stills needs further attention before we can incorporate it into ADL. The main problem is defining the related semantics. For example, when we replicate an activity it is yet unclear how we should replicate the communication structure. If, for instance, activities A and B communicate by means of a synchronous channel, does this mean that replication of A into activities $A_1, \ldots, A_n$ should also yield n replicated synchronous channels (and corresponding gates at B)? Replication is going to be incorporated in ADL, but again, we have deferred the matter until a later stage.

4 A Design Entry System: implementation of ADL

The implementation of ADL/1 forms part of the so-called *Design Entry System*, or DES for short. The DES basically consists of the following three components:

- An implementation of ADL/1 in the form of a graphical editing system by which only syntactically correct ADL designs can be made.
- An implementation of a graphical version of the INMOS *Network Description Language* (NDL) [5], by which a target transputer system can be configured for a specific application.
- A transformation system which generates the necessary configuration files for software and hardware components, and the mapping between them, as well as skeletal code for the application described in ADL/1.

The graphical version of the INMOS NDL, referred to as NDL/Graph is implemented quite similar to ADL. Conceptually, it is much simpler than the ADL due to the relative straightforward semantics of the INMOS NDL. It is beyond the scope of this paper to discuss in detail how we have actually implemented NDL/Graph, but reference to its implementation will be made when discussing the ADL implementation below.

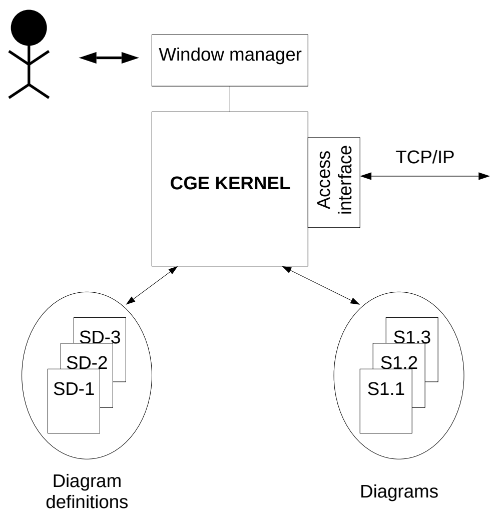

Figure 4: Global architecture of a customized version of the CGE.

4.1 Global system architecture

4.1.1 The CGE: a configurable graphical editor

A component that is paramount in the DES from an end user's point of view is the *Configurable Graphical Editor* (CGE) developed at TNO-TPD [10]. Basically, the CGE is a 2D customizable multiwindow graphical editor that can be adapted for a wide range of diagram techniques. The only restriction is that the logic structure of diagrams belonging to a diagram technique can be mapped on a network (graph). The CGE customizes itself after it has been provided with a correct *Diagram Technique Definition File* (DTDF). This diagram technique definition file should be constructed precisely once for all for each of the diagram techniques that should be supported, and can then be used repeatedly. In the case of our DES, two diagram techniques will be supported: one for the ADL, and one for NDL/Graph.

When customized for a specific application, the CGE assists the user in constructing diagrams in a such a way that only syntactically correct diagrams can be constructed. The CGE considers only the structure of a diagram: no attempt is made to include its semantics. When the CGE is used within a technical design environment additional tools should take care of that aspect. This is further discussed below, but for now it is important to note that the CGE has a special interface layer which allows communication with the outside world. In this way, not only can diagrams be manipulated by other applications, more important is that we can add functionality to a CGE-based support tool by means of independent components. The architecture of a customized CGE is depicted in Figure 4.

Before the CGE can be used by the end-user, a definition file must be loaded. This file contains a description of all symbols and rules associated with a specific diagram technique, using a special definition language, the *Diagram Technique Definition Language* (DTDL). A diagram technique is defined by describing the graphic symbols, the possible manipulation on these symbols, the possible interconnections between the symbols and the constraints with respect to the structure of a diagram. All these aspects are integrated into the DTDL.

4.1.2 Adding functionality

An important feature of the CGE is its interface layer by which external applications can communicate with the editor. This interface layer is based on the TCP/IP protocol and provides a high degree of flexibility for communicating with technique-specific applications that capture non-graphical functionality. To this aim, an alternative version of a diagram as constructed with the CGE should be constructed. This alternative version is nothing but a datastructure capturing the same information as the original diagram but which can be accessed by external applications. The datastructure is generated automatically by what we refer to as an *Abstract Data Type* (ADT) builder.

Note how, in our case, the ADL/ADT builder actually creates *instances* of an abstract data type. Each instance corresponds to a specific diagram representing a model of the application expressed in ADL. Of course, in order to create a datastructure, it is necessary that a *definition* of the diagram technique is also available. This definition is actually the abstract data type mentioned before. We shall return to this issue below, but for now it is important to note that this abstract data type itself is also automatically generated by the CGE.

Returning to the DES, we are now able to expose its entire global architecture. Two main subarchitectures can be distinguished as depicted in Figure 5: one that deals with graphically configuring a target transputer system, and one that handles the design of applications using ADL. Similar to the ADL subsystem, the NDL subsystem generates instances of the so-called NDL Abstract Data Type. These datastructures are used by the ADL/ADT builder to relate processes as described in an ADL design to transputers, and to add this mapping information in an ADL/ADT instance. Mapping information is assumed to be provided by hand when designing an application in ADL, given a description of the target hardware expressed in NDL/Graph.

Using the ADL/ADT instances as input, several additional tools are invoked which eventually result in a set of configuration files by which the application can be loaded onto the target network, as well as a set of files containing skeletal code for each of the processes to be executed.

4.2 The development approach

Development of the DES would be an extremely cumbersome task if it was to be done from scratch. For example, if transformation tools were to be developed that directly transformed ADL diagrams into configuration files, it is not hard to imagine that much redundant work would need to be done if it was decided to use a completely different network configuration language. We have been frequently confronted with this situation when installing the CGE at one of our customer's sites. In order to ease the development of CGE-based CASE environments, it is essential that a more sophisticated approach is followed. As we have already briefly mentioned above, many datastructures are automatically generated by our system. In this subsection we take a closer look at the way we actually generate a CGE-based CASE environment in a semi-automated fashion.

4.2.1 Modeling the CASE environment

Essential in our approach is the construction of models (at different levels of abstraction) of what actually constitutes a CGE-based CASE environment. To this aim, we have developed

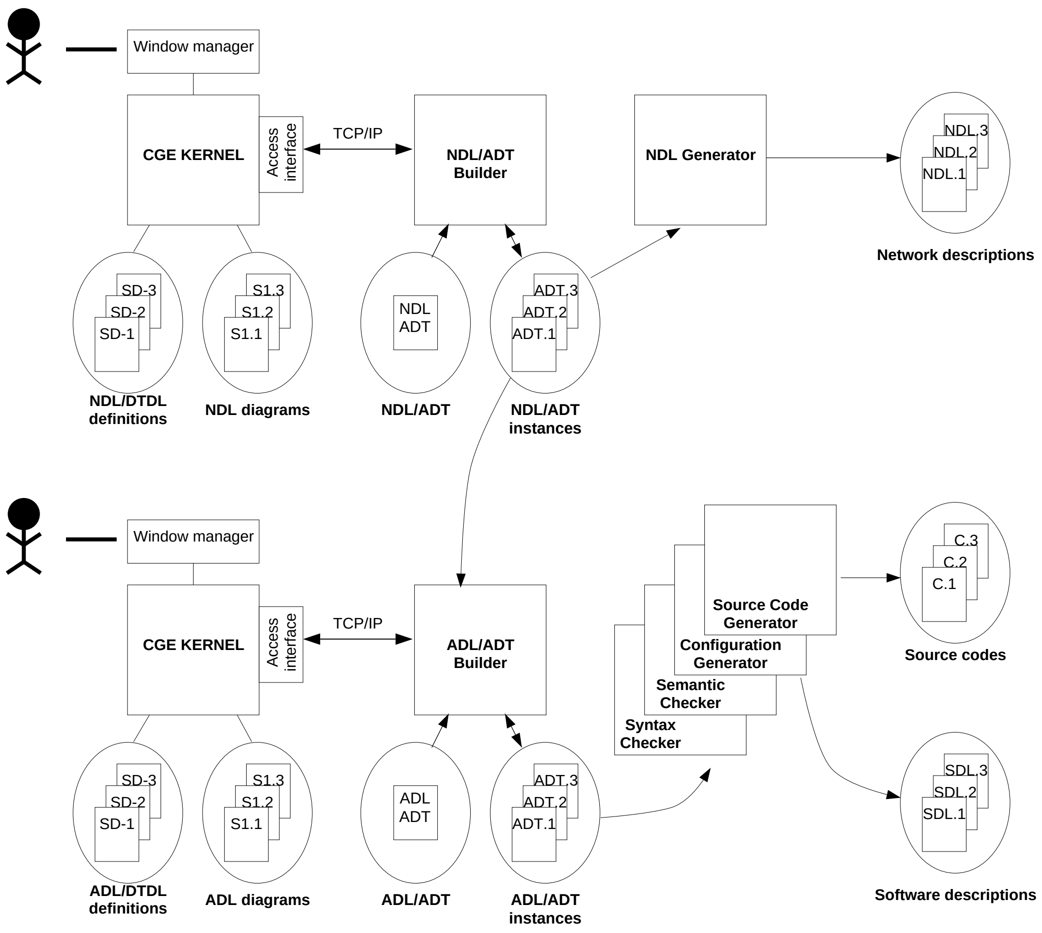

Figure 5: The global architecture of the Design Entry System.

the following semantic data models expressed in NIAM[1].

- *The NIAM Schema.* This model allows us to bootstrap a customized CGE-based environment. It describes our definition of NIAM, again expressed in NIAM. After instantiating this schema, we then have the basic means to define, create, and manipulate models expressed in NIAM.
- *The DTDL Schema.* This model forms a definition of the Diagram Technique Definition Language. Any instance of this schema corresponds to an actual definition of a diagram technique, including all sorts of information related to the graphical representation of diagrams.

The important thing to note here is that all these models have been expressed in the same formalism, namely NIAM. This is an important issue for this common description language allows us to construct transformations to other models expressed in the same formalism rather easily. Our problem thus essentially reduces to *model transformations*.

For example, in order to generate skeletal code from ADL designs, we have additionally constructed the following models also expressed in NIAM:

- *The ADL Schema.* This is nothing else but a language definition of ADL expressed in NIAM. Obviously, an instance of this schema corresponds to an actual design expressed in ADL.
- *The Code Schema.* This is a more or less general model of imperative programming languages, capturing the semantics of declarations for programs, modules, functions, arguments, variables, etc. Using this model, we are capable of at least expressing the declarative parts of an imperative program[2].

These models allow us to easily create a fully customized CGE-based environment in a partially automated fashion, as is discussed next.

4.2.2 Semi-automated construction of the DES

Concentrating on the construction of the DES subarchitecture that supports development of ADL designs, we have implemented ADL/1 by proceeding according to the following steps.

Step 1. We start with defining the concepts of ADL using the DTDL. This step results in a *textual* definition of ADL expressed in DTDL. This definition of ADL is needed only to customize the CGE so that we can actually draw diagrams. Not surprisingly, this definition is entirely aimed at the concrete *graphical* syntax of ADL.

[1]NIAM is a formal method based on modeling binary relationships, comparable to extended versions of the Entity-Relationship model. The interested reader is referred to [8].

[2]We note here that source code generation by the DES initially only covers declarations, and sections for initialization and finalization.

Step 2. The second step is entirely automated by means of the ADT builder, already mentioned in Subsection 4.1.2. It involves the following activities:

2a. The builder starts with instantiating a NIAM model of NIAM resulting in an internal datastructure that allows us to define, create, and subsequently manipulate models expressed in NIAM.

2b. It then continues with loading our NIAM model of the DTDL definition, or, in other words, *instantiating* the previously generated datastructure for NIAM models. At this point we are now capable of creating and manipulating diagram technique *definitions* expressed in DTDL.

2c. Finally, our textual definition of ADL from the first step is parsed by the generator and created as an instance of our NIAM model of the DTDL. In other words, the ADT generator fills in the datastructure that resulted from step 2b with the specific ADL diagram technique *definition*, now allowing us to actually create and manipulate ADL diagrams. The result is what we have previously referred to as the ADL/ADT.

Note how Steps 2a and 2b correspond to generating a database for storing and manipulating *models* expressed in some general *model definition language* (in our case DTDL). Step 2c then corresponds to customizing such a database so that models expressed in a specific formalism can be created and manipulated (in our case ADL). Returning to the previously mentioned interface layer of the CGE, we can now state that this is, in fact, nothing but a data definition and manipulation language for model bases, albeit rather primitive.

Step 3. This step involves explicitly relating the ADL definition expressed in DTDL to the one expressed in NIAM and consists of two activities. In the first place, we manually load the NIAM model of ADL (previously referred to as the ADL Schema) into the system, or, in other words, we instantiate yet another NIAM model of NIAM similar to what was done in Step 2b. Furthermore, we develop a parser that transforms any ADL diagram description in terms of DTDL to one expressed in NIAM according to the NIAM model of ADL.

Step 4. The fourth step consists of manually loading the Code Model (which was also expressed in NIAM) into the system.

Step 5. Finally, we concentrate on the development of a component which transforms an ADL diagram expressed as an instance of a NIAM model of ADL, into an instance of the Code Schema. This step, so far, is the most time-consuming in the construction of a CGE-based CASE environment. Also, this fifth step deals with developing the actual source code generators for C, Occam, or INMOS NDL.

Our development process is depicted in Figure 6. Summarizing, by using a common description language (in our case NIAM), we have been able to actually reduce the effort of developing a CGE-based CASE environment to the development of a transformation system that acts on instances of the ADL/ADT, and to construct code generators for the actual target languages C, Occam, and INMOS NDL.

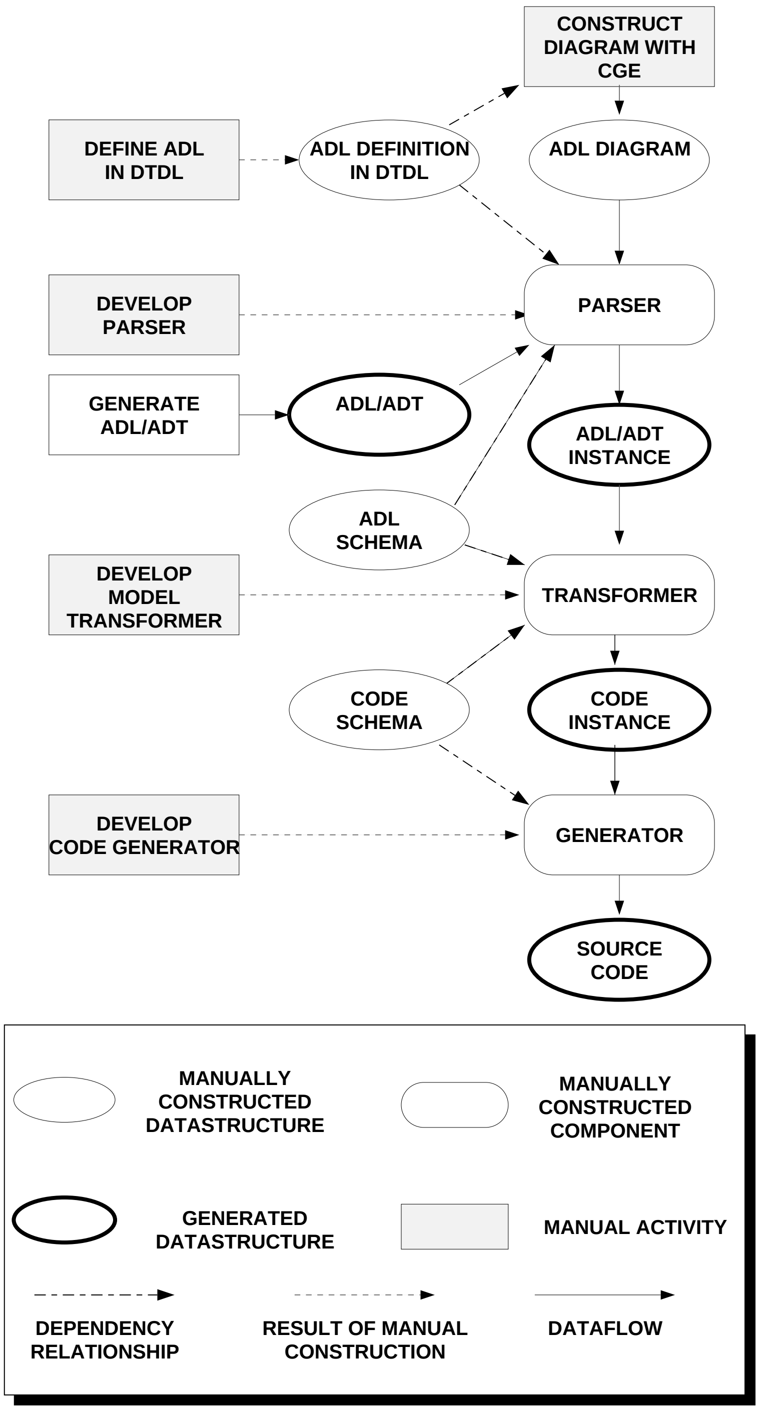

Figure 6: The DES development process.

5 Current results and future work

Due to the fact that the application developers in the Hamlet project are in need of *any* support they can get during the design phase of their applications, we have decided to follow an incremental approach with respect to developing the DES. At present, a graphical editor for ADL (as well as NDL/Graph) has been completed and will be released to the application developers soon. Although an initial version was ready within only a few days after freezing the definition of ADL/1, we had to ensure upwards compatibility with future releases. This requires a careful design of the DTDL description of ADL/1.

We are currently working on the automated generation of source code and configuration information thereby initially taking C, enhanced with Parsytec's RTSM communication library [4], as our target programming language, and the INMOS NDL as our target configuration language. At the moment of this writing, a very rudimentary source code generator is available, as well as an INMOS NDL generator. When considering the fact that our actual implementation work started in the beginning of this year with a team consisting of one senior and one junior software engineer, we feel confident that our development approach as sketched above is indeed a feasible one for rapidly developing a customized CASE environment. It is also clear that this approach allows us to concentrate on the essence: the transformation of ADL designs to source code and configuration files.

But much work is still be done. As mentioned, we need to enhance ADL/1 in such a way that state-transition diagrams, as well as replication are supported. Also, much effort should be put into the development of the actual transformation system and code generators. Only if we are capable of producing code that can indeed be itself embedded in industrial applications, may we call our participation successful.

Acknowledgements

This work has been conducted as part of the ESPRIT project Hamlet (P6290), and is partially funded by the Commission of the European Communities. We also particularly thank Jaap Gordijn from TNO-TPD for carefully reading the manuscript.

References

[1] M.J. Bach and S.J. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Technical Journal*, 63(8, part 2):1733–1749, October 1984.

[2] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *Computing Surveys*, 21(3):261–322, 1989.

[3] HAMLET. Application Requirements. HAMLET Technical Report, AEG Electrocom, Konstanz, Germany, September 1992.

[4] HAMLET. RTSM Description and Preliminary User Manual. HAMLET Technical Report, Parsytec Industriesysteme, Aachen, Germany, January 1993.

[5] N. Haydock. NDL Hardware Configuration Language Reference Manual. Internal SW-0308-10, INMOS Limited, June 1992.

[6] I. Jacobson. *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison-Wesley, 1992.

[7] A.H. Karp. Programming for Parallelism. *Computer*, pages 43–57, May 1987.

[8] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design, A Fact Oriented Approach.* Prentice-Hall, 1989.

[9] H.R. Simpson. The Mascot Method. *IEE Software Engineering Journal*, 1(3):103–120, May 1986.

[10] T. Vogel. Configurable Graphical Editor, Users Guide. Technical Report 91 ITI 382, TNO Institute of Applied Computer Science, Delft, February 1991.

[11] P.T. Ward. The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, SE-12(2):198–210, 1986.

[12] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*, volume I, II & III of *Yourdon Computing Series.* Yourdon Press, Englewood Cliffs, N.J., 1985.

[13] E. Yourdon and L.L. Constatine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Prentice-Hall, Englewood Cliffs, N.J., 1979.