Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

R. van Liere, P.J.W. ten Hagen

Resource management in DICE

CWI BIBLIOTHEEK
3 0054 00063 4890

Department of Computer Science Report CS-R8746 October

Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

69K34, 69K35, 69K36

Copyright © Stichting Mathematisch Centrum, Amsterdam

Resource Management in DICE

R. van Liere, P. J. W. ten Hagen

Department of Interactive Systems
Center for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

A framework is presented for integrating a general resource management facility into the dialogue cell language. It is shown that, by using resources as the basis for input and output, the coupling of input and output at the physical device level can be achieved. By integrating the resource manager in the dialogue cell language, correlations between higher level input and output can be defined and maintained.

Context sensitive resource rules are defined as extensions to the corresponding activation rules of dialogue cells themselves. By applying various inheritance mechanisms a resource specification can be done virtually. The dialogue run-time system dynamically binds these virtual specifications with physical devices.

The resource manager is implemented by augmenting the dialogue grammar with specific resource information. In this way a potentially ambiguous dialogue can be unambiguously parsed. An $O(n \log n)$-time algorithm, with n indicating the number of overlapping windows, is given which detects ambiguous resource configurations.

CR Categories: I.3.4 [**Computer graphics**] : Graphics utilities- *application and graphics packages*; I.3.5 [**Computer graphics**]: Computational geometry and object modeling- *geometric algorithms*; I.3.6 [**Computer graphics**]: Methodology and techniques- *device independence, languages*

Key Words & Phrases: User Interface Management Systems, dialogue languages, resource management, window management

Note: This report will be submitted for publication elsewhere.

1. Introduction.

The dialogue cell specification method, or simply DICE, has successfully been used as a tool for designing high quality graphical user interfaces. With this method a user interface designer defines the syntax of a dialogue by specifying a context-free like grammar. A context-free grammar is capable of defining a large part of the syntax of a typical dialogue language, and the existence of a variety of syntax-directed parsing techniques has facilitated the construction of efficient parsers for such syntax definitions.
This form of definition has adequate power to define the syntax of dialogue languages.

Report CS-R8746
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Unfortunately, every context-free dialogue language will need some syntax extensions which are context-sensitive. Therefore, a dialogue language syntax definition consists of two parts:

- a context-free grammar which defines the sentence structure of the language.
- a set of additional rules which impose context constraints on the language.

Each context-sensitive rule may require some relation to hold among certain elements occurring in the dialogue specification. For instance, DICE allows a particular window to be shared by two dialogue cells. In which screen area the two cells actually are displayed is a part of the dialogue syntax. Therefore, this requirement can be represented as a context-sensitive rule which allows the definition of a window to depend on a related dialogue cell's window.

Context sensitive rules are conventionally expressed in an *ad hoc* fashion, which makes it a difficult task for the designer to apply them correctly in a dialogue specification. This is in particular true for larger dialogue specifications; i.e. specifications consisting of hundreds of dialogue cells in which numerous cells come from predefined libraries. In this paper we are concerned with how the dialogue cell system manages the declaration and application of resources, and the impact resource management has on the execution of a dialogue. We also show that, by using techniques familiar to parsing attributed grammars, the implementation of resource management can be combined with other elements of the dialogue specification into one formalism.

Before going into details of how the DICE system treats context sensitive resource rules internally, it is appropriate to describe these rules form a user point of view. The basic assumption is that at any time during the dialogue the user has a choice between more than one type of input entity. For instance, the user may have the freedom to decide whether to input text, select from a menu or to input geometrical data through a mouse. This variety of choice may be caused by the following situations:

- there is more than one input component item being constructed.
- for the same compound item the order of providing the constituents is immaterial.
- there is a choice between alternatives of different types.

In each of these situations the possibility arises that there is more than one 'request' for the same input token; for instance, when the mouse is used for more than one purpose. This confronts both the system and the user with an ambiguous situation since it is not known which 'request' will be given the input token. By adding additional context-sensitive syntax to the syntax rule it is possible to make clear to the user which token goes where. The basis for adding this extra syntax is to dynamically differentiate the mechanisms which provide these tokens. The differentiation is provided by the resource manager who controls all output and input mechanisms including the screen management. Differentiation can thus result from different output presentations, different input devices or a combination of both. As we will see, this implies that the resource manager also handles the window management. For this reason, it is our firm believe that window managers should be incorporated in resource managers rather than only provide a variety of windowing functions.

Within the dialogue system itself every dialogue cell has associated with it a *resource*; i.e. a description of the physical resources that are assigned to the corresponding dialogue cell. The dialogue cell's resource provides the link which allows a device independent dialogue cell specification to be mapped onto physical resources of the workstation.
Resource specification is done in terms of virtual device classes which are (partial) static descriptions of device independent resources. The collection of resource specifications results in an underlying *resource model.* There are two advantages for using such a resource specification scheme. Firstly, since the activation environment of a dialogue cell is statically unknown, a dialogue designer does not know on which part of the display a dialogue cell is activated. This resource specification scheme allows the dialogue designer to specify resources relatively with regard to its parent resource. Secondly, this resource specification scheme allows the designer to partially define a resource. By inheritance, the activation environment information of a parent cell can be used to complete the resource specification.
For instance, the window component of the resource can be specified in terms of the parent dialogue cell's resource. This (partial) resource specification is filled in at run time by inheriting the parent cell's resource.

The resource manager will take information from the resource model to make decisions concerning the allocation (and reallocation) of resources. Basically, the responsibilities of the resource manager are twofold :

- To ensure that various predefined language dependent criteria, in particular those which disallow so-called ambiguous situations, are obeyed whenever a resource request is granted. Ambiguous situations occur in parallel systems when more than one reaction to a specific user input is possible. Clearly, ambiguities should be avoided since they leave the user in a confused state.

- To ensure that the resource model is always in a consistent state. Operations on the resource model do not only include those when resource specifications are elaborated (i.e; resource allocation), but also those done by the user when the window layout is changed.
  For instance, if a user moves a window to another area of the display, all other related windows must also be moved. Similarly, if a window is reshaped, all other related windows must also be reshaped accordingly.

The context-sensitive rules applying to resource management are implemented as functions which evaluate attributes in the so-called *attributed dialogue grammars.* In general, attributed dialogue grammars are automatically derived by augmenting the original dialogue grammars with various predefined attributes. Using attributed dialogue grammars allows semantical issues of the resource manager to be validated as the evaluation of attributes in the attributed dialogue grammar. This has two advantages over the more traditional procedural approach: (i) the semantics is expressed applicatively. Propagation of attributes through the derivation tree is implicit in the formalism and need not be specified imperatively on a case-by-case basis and (ii) the specification is modular; the arguments to each semantic function are local to one production.

1.1. Format of this paper.

Section 2 first gives an overview of dialogue cell languages and informally states the parsing problem inherent to these languages. This leads to the definition of a number of simple criteria that must be obeyed in order to be able to parse these languages.
Section 3 gives a description of the underlying resource model and shows in which way the resource model must be constrained in order to meet the criteria laid upon the system by the parser.
Section 4 gives an overview of how such a resource model and the resulting (static as well as dynamic) operations on it can be implemented.
Appendix 1 shows how, given a dialogue cell grammar, a context-free grammar can be constructed. Appendix 2 gives an overview of the definition of attributed dialogue grammars. Appendix 3 reviews the logical input device model.

1.2. Related work.

Specifying user interfaces by regular / context-free grammars and using parsing techniques to execute the resulting dialogues is not new. Pioneers in this area have been Newman ([12]) and Olsen ([13]) who were the first to build such systems.
Dialogue cell type grammars have been studied by van den Bos ([20]), Holt *et. al.* ([7]) and Lewi ([2]); the last two studies have been in the field of specification languages suited for the construction of compilers. Van den Bos was one of the first to recognize the merits of non-determinism in dialogue languages.
Defining the semantics of a context-free language by means of an attribute grammar was introduced by Knuth ([9], [10]). The first paper considers various criteria which ensure that attributes in a grammar are correctly evaluated. Considerable research has been done in the area of attribute grammars and (re)evaluation of attributes. Recently, novel algorithms for incremental evaluation of attributes have been developed by Demers, Reps and Teitelbaum ([3]) in the domain of syntax directed editing. These algorithms have also been used as a basis for error recovery schemes in a user interface management system ([8]).
To our knowledge, using incremental evaluation techniques have not been applied to resource management. Introductions to dialogue cells can be found in ([1] , [19]). The impact that non-determinism has on feedback is given in ([15] , [14]). Finally, ([18]) presents dialogue cells in a somewhat more fundamental way by defining a general model for graphical interaction.

2. Dialogue cell languages.

A dialogue cell grammar is characterized as the 4-tuple:

$$G = <\Sigma_N, \Sigma_T, P, S>$$

where Σ_N and Σ_T are the disjoint finite sets of *dialogue cell symbols* (= non-terminals) and *basic dialogue cell symbols* (= terminals) respectively, P is the set of symbol expressions and $S \in \Sigma_N$ is the root cell symbol.

Σ_T is the union of the basic input symbol set and the combining operator set (denoted as $\Sigma_{T_I} \cup \Sigma_{T_O}$) in which

$$\Sigma_{T_I} = \{ \textit{Locator}, \textit{String}, \textit{Choice}, \textit{Pick}, \textit{Valuator} \}$$

is the set of terminal symbols, and

$$\Sigma_{T_O} = \{ ;, \wedge, \vee, \textbf{case}, \star, + \}$$

is the set of combining operators.

Symbol expressions are written in the form:

$$A \rightarrow \theta\,(E_1, \;\cdots\;, E_n)$$

where A is the dialogue cell symbol; θ is a combining operator and E_i are the symbol subexpressions which are, analogous to expressions in conventional programming languages, factorized into simpler forms until eventually a so-called subcell is obtained. Subcells belong to the the set $\Sigma_N \bigcup \Sigma_{T_I}$.

Operators define the order in which the subcell results are to be accepted. The sequence operator, ";", indicates that the order in which subcell results are accepted is sequential. The parallel operators, "$\vee$" and "$\wedge$", allow subcell results to be accepted in any order. In the case of the "$\vee$" operator, only one of results is necessary whereas the "$\wedge$" operator requires that all results must be accepted before the complete subexpression is reduced. The iterator operators "+" and "$\star$" allow iteration. In this case, the resulting value of the subcell determines the stop condition. Similarly, the **case** operator allows for a branch depending on the result of the first operand. The iterator and case operators are special cases of a more general scheme in which result values of symbols can influence shift / reduce parsing decisions.

All dialogue cells are strongly typed in the sense that the use of result values must correspond to the definition. In particular, basic dialogue cells deliver (in some workstation dependent way) result values of predefined types; i.e. *Locator*, *Choice*, *Pick*, *Valuator*, *String*. Higher order dialogue cells can return values of arbitrary user defined types.

Finally, user defined semantic actions can be associated to each symbol subexpression in the form of so-called echo and value trigger actions. These are, analogous to syntax directed translation schemes, executed each time a symbol subexpression is parsed. Echo trigger actions are denoted as

$$A_e\ (E_i) : erule_{i1},\ erule_{i2},\ ...,\ erule_{in}$$

In this case, A is the name of the dialogue cell, E_i is the trigger expression and $erule_i$ are the echo mapping rules. In this way arbitrary (even application dependent) feedback can be given. Similarly,

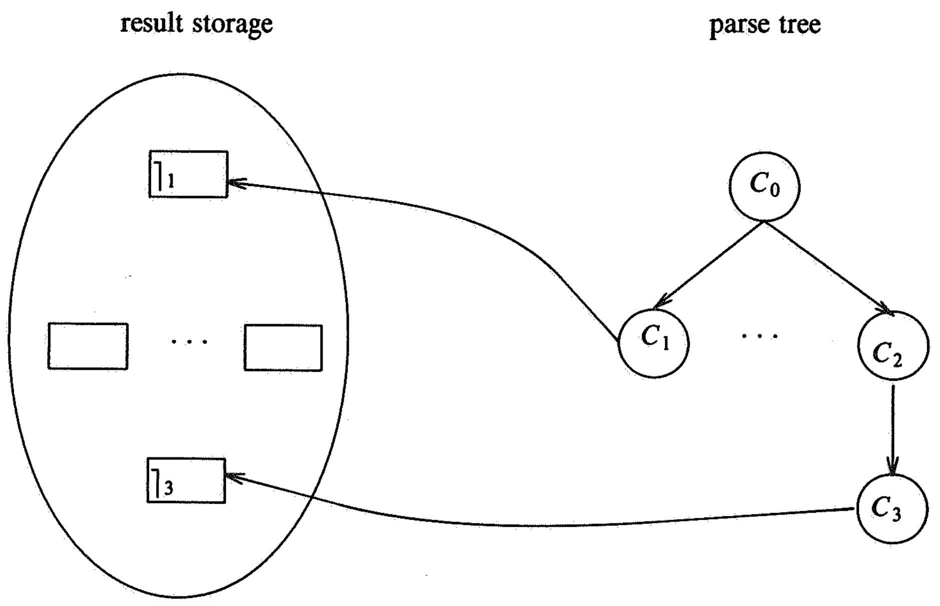
$$A_v\ (E_i) : vrule_{i1},\ vrule_{i2},\ ...,\ vrule_{in}$$

denotes a value trigger action. Specifying value actions allow input values to be interpreted as soon as they are consumed by a cell.

Triggering value actions followed by echo actions in this way provides the dialogue programmer with a mechanism that guarantees a certain synchronization between what the user sees on the screen and how the the dialogue has interpreted the internal values; i.e.

"what you see is what you get". Alternatively, triggering an echo action followed by value action provides the dialogue programmer with a mechanism that allows to visualize the result of an action even before it has been executed internally.

Conceptually, the dialogue is executed as follows. Each basic dialogue cell is simulated by a physical input device. After reading the physical input device triggered by the user, the corresponding basic dialogue cell produces a result symbol. The produced result is input for another cell, which, in turn, will eventually produce a result. This process continues until the root cell symbol produces a result. Each time a result is produced, it is dumped in a storage pool. A scheduler will examine the results in the storage pool to determine which result must be processed. A scheduler is necessary since dialogue cells run in parallel. † Result passing is depicted in the following figure. Since the parse tree defines the flow of the dialogue, the source and destination of each cell result is known to the scheduler.



In the context of resource management it is important to note that, in order to produce a result, a dialogue cell must first be activated. Activation consists of the initialization of the cell plus (depending in the mode) of the activation of the subcells. In particular, all resources that are needed by a dialogue cell are requested at initialization time. Using such a synchronous activation scheme in conjunction with the resource model, deadlock situations are easily detected.

Once activated, dialogue cells can execute either synchronously or asynchronously. Considerable flexibility is achieved by allowing cells to be activated asynchronously

† The overhead required to do scheduling can be constrained by introducing processing priorities in the semantics of the language (see [17]). Such a scheduler can be viewed as a generalization of a lexical analyzer. This execution scheme has many similarities with event-driven systems, such as [5] and [6].

since it allows a user to determine the moment of giving an input. It must, however, be emphasized that asynchronous activation differs significantly from the non-deterministic operators in the symbol expressions. Operators within a symbol expression determine the *order* in which the results of the subcells are parsed whereas the activation mode merely determines the *moment* in which a subcell may start producing a result.

For every dialogue language there exists an equivalent context-free language. Appendix 1 gives an algorithm which converts a dialogue grammar into a context-free grammar. Unfortunately, the process of parsing a particular dialogue differs significantly from parsing a string belonging to a context-free language. The problem of parsing a dialogue is the subject of the next section.

2.1. The parsing problem.

The major differences between parsing a dialogue and parsing a string in a context-free language are summarized as:

- A dialogue requires a $LR\,(0)$ type parser since every trigger action must be executed as soon as an input is received; i.e. a user cannot be expected to give k inputs before getting feedback from the first input, as is the case of $LR\,(k)$, $k \geqslant 1$ parsers !
- In contrast to conventional $LR\,(k)$, $k \geqslant 0$ parsers, dialogue parsers obtain tokens from multiple input streams. This is due to the various parallel operators and the asynchronous activation modes. Hence, the dialogue parser is *simultaneously* parsing multiple input expressions. Parallelism results in extra *ambiguous situations* since it cannot be determined to which stream a particular input token belongs. These situations cannot be resolved by only using techniques familiar to conventional $LR\,(k)$, $k \geqslant 0$ parsers.

  Ambiguous situations can be brought down to the following two canonical forms :

- Symbol expressions which have the form :

  $$A \rightarrow \theta\,(B,\, C) \qquad \theta \in \{\wedge,\, \vee\},\ C \in \Sigma_{T_I} \tag{1}$$

  $$B \rightarrow C \tag{2}$$

  These two symbol expressions will cause an ambiguous situation when the device belonging to basic dialogue cell C is triggered.

- The symbol expression of a dialogue cell has the form :

  $$A \rightarrow \theta\,(B,\, B) \qquad \theta \in \{;,\, \mathbf{case}\},\ B \in \Sigma_{T_I} \tag{1}$$

  This unambiguous symbol expression causes ambiguous situations when subcell B is activated asynchronously.

For instance, assume two dialogue cells have the symbol expressions

$$A \rightarrow \wedge\,(B,\, \textit{locator})\ ,\ \ B \rightarrow \wedge\,(\textit{locator},\, \textit{locator})$$

with *locator* $\in \Sigma_T$. In this case, three dialogue cells with the name *locator* are simultaneously active. When a token with the type *locator* is received by the parser, it cannot be determined which input stream is implied.

We now, informally, summarize the approach taken by the dialogue cell parser to parse a dialogue. Conceptually, during the activation of *every* subcell, a unique input stream is created in which the subcell stores its results. Since each input stream is unique, the correspondence between a subcell and it's parent cell is also unique. The input stream is removed as soon as the subcell is deactivated. Each input stream is typed with the type indicated in the specification the subcell and is uniquely labeled. By augmenting tokens with the input stream identification, the parser can uniquely determine the corresponding production rule. This scheme avoids the ambiguous situations stated above since the correspondence between the input stream and the dialogue cell that processes the token is now unique.

This can be formalized somewhat by defining an input token to be a triple, denoted as $<$*type, value, id*$>$, consisting of a dialogue cell result type, dialogue cell result value and a stream identification. For every token that is ready to be processed, the following relation must hold (we use the notation $X\,[Y_i]$ to indicate the projection of element Y_i from the n-tuple $X = <Y_1, \cdots, Y_n>$) :

$$\forall\ token_i,\ token_j,\ i \neq j : token_i[id] \neq token_j[id] \qquad (1)$$

Associated with each dialogue cell A is a set of open input streams, denoted as $InStream_A = \{i \mid StreamOf\,(A, i) = \textbf{true}\}$ in which the predicate

$$StreamOf : CellName \times Stream \rightarrow Boolean$$

determines if an active dialogue cell contains a particular open input stream. The scheduling function for active dialogue cell A is denoted as:

$$schedule_A\ (\{token_i \mid \bigcup_i\ token_i\}) = token_j$$

i.e. return a token for dialogue cell A, given the set of possible input tokens. The scheduled token will obey the relation

$$token_i\ [id] \in InStream_A$$

At the lowest level of the derivation tree the workstation provides a predefined number of typed input streams, corresponding to the result types of the basic dialogue cells. This is denoted as the set :

$$BasicStreams = \{i \mid StreamOf\,(A, i) = \textbf{true} \wedge A \in \Sigma_{T_I}\}$$

The cardinality of *BasicStreams* is bounded by a predefined, workstation dependent number:

$$\mid BasicStreams \mid < BasicStreams_{IsMax} \qquad (2)$$

in which $BasicStreams_{IsMax}$ is the maximum number of input streams provided by the workstation.

Implementing such a scheme is fairly straightforward. The strategy taken by the dialogue cell system is to augment each dialogue cell production rule with a set of predefined *attributes.* Attributed dialogue grammars are very similar to attributed context-free grammars † in the sense that functions are provided with each attribute that associate a value to that attribute. Moreover, as in the case in conventional attribute grammars, attributes can also be *inherited* and *synthesized.* The general idea of using attributes is that various context sensitive conditions at one level in the parse tree can be used at another level. New context conditions can be determined by traversing the tree and (re)evaluating the attribute values at that particular node. As is the case of attribute grammars, a major problem is how attributes can be efficiently (re)evaluated. Attribute reevaluation is achieved in the dialogue cell system by constraining the "scope" of the attributes in a language dependent way. In this way, only (small) portions of the tree is traversed in order to reevaluate the corresponding attribute values.

3. Resources.

In the previous section we have shown which restrictions are implied on the execution of a dialogue in order to allow a non-ambiguous parse. We now define a scheme for managing resources which, when applied to dialogue grammars and languages, serves as an implementation of these restrictions.

A resource is a logical description of the physical hardware resources, consisting of a window description, an input stream description and a priority. Associated with *every* dialogue cell is a resource. The exact value of the resource is determined at activation time of the dialogue cell and cannot be changed during the lifetime of the cell. Since the order of activation is defined by the semantics of the dialogue cell system, it is possible to define predicates that avoid deadlock situations. The notation used is :

$$r = <w, i, p>$$

where r is the name of the resource, w the window description, i the input stream description, and p a priority.

The total screen space can, through an appropriate overlap strategy, become a much larger virtual screen space. Each of these virtual screen spaces are defined to be a window. How window descriptions are eventually mapped onto the physical screen space depends on scope rules that are laid upon the instantiation of a window (these rules are discussed in section 4.1.1).

Input streams are typed by the values that are returned from the dialogue cell. At the lowest level of the cell hierarchy input streams are simulated by logical input devices † by using logical input class types as primitive input stream types. Higher level input stream types are defined as the composition of lower level streams types. The input stream description contains both the type of the input stream as well as the

† A summary of attributed dialogue grammars and the used notation for these is given in Appendix 2.
† A summary of the concept of logical input devices is given in appendix three.

identification of the stream; i.e. $i = <\mathit{InputStreamType},\ \mathit{InputStreamId}>$.

Basically, there are two aspects relevant to resource management: *resource organization* and *resource allocation / deallocation*. We first introduce both of these aspects before going into each of them in more detail.

- Resource organization follows directly from the specification of resources in the dialogue cell *and* the environment of the parent dialogue cell's resource. Resource specification is done in so-called virtual window and input device classes. A virtual window class is a static description of a set of windows in a virtual coordinate space. Similarly, a virtual input device class is a description of a set of logical input devices. Classes are instantiated during run-time, by mapping the virtual spaces onto the physical spaces of parent resources. This results in a *resource model.*
- Resource allocation is done in run-time by binding resource descriptions onto physical resources; i.e. a window onto a screen portion and (at the lowest level of the cell hierarchy) a virtual input device onto a particular logical input device. There are two issues that must be considered with respect to resource allocation :
  - ⊕ Fast changing contexts, which are typical in interaction, require dynamic redistribution of resources. By constraining class definitions, the ambiguity resolution algorithm should only have to consider competing resources that are *local* to the requested resource. In particular, by defining the resource model to be strictly hierarchical, resource allocation requires only the traversing of one branch in the class hierarchy.
  - ⊕ The user must be given the ability to dynamically overrule decisions made by the resource allocation scheme. For instance, the window layout may be altered or the mapping of virtual to physical input devices may be altered. However, when the user alters a resource configuration, it must be assured that the resource ambiguity criteria are not violated. Note that, by altering the window layout, not only will various logical input device echo areas be altered, but also all windows which depend on the altered window.

The restrictions laid upon a dialogue in the section 2.1 can be translated into criteria that must be satisfied by a resource. Intuitively, this can be justified by considering the augmented token identification to be a resource. Showing that a resource is unique will then be sufficient to guarantee that the token produced by the dialogue cell can be uniquely identified. A resource, r_i, is defined to be unique if one of the following conditions is satisfied :

(i) $\forall r_j,\ i \neq j :\ r_i[w] \cap r_j[w] = 0$

(ii) $\forall r_j,\ i \neq j :\ (r_i[w] \cap r_j[w] \neq 0) \wedge (r_i[p] = r_j[p]) \Rightarrow (r_i[i] \neq r_j[i])$

(iii) $\forall r_j,\ i \neq j :\ (r_i[w] \cap r_j[w] \neq 0) \wedge (r_i[i] = r_j[i]) \Rightarrow (r_i[p] \neq r_j[p])$

Condition (i) assures resource uniqueness if the windows are non-overlapping. Condition (ii) assures that two resource are unique if they have overlapping windows and identical priorities but have different input descriptions. Condition (iii) states that if two resources have overlapping windows and identical input descriptors, then they must have different priorities. By including a window description in the resource description allows the non-ambiguity criteria to be applied to *each* window.

For the logical input streams the following relation holds:

$$| Is_W | < BasicStreams_{IsMax}$$

where: $Is_W = IsLoc_W + IsPick_W + IsChoice_W + IsString_W + IsValuator_W$ and $W \in \{ w_i \mid WindowExists\ (w_i) = \textbf{true}\}$; i.e. the number of open basic input streams in one window is bounded by a workstation dependent maximum. The total number of open basic input streams is the sum of the open basic input streams for each class.

4. Implementation.

Implementing the concepts of resources, classes and inheritance is done by merging them into the parsing process of dialogue languages and treating them as attributes in the corresponding augmented dialogue grammar. Resource allocation is expressed in terms of the evaluation of attributes during the activation of a dialogue cell. Redistribution of resources can be expressed as the reevaluation of the corresponding attributes by traversing the derivation tree and marking the effected attributes and their dependencies. Only these marked attributes will then have to be reevaluated. This process is similar to the ones described by Reps [3] and by Hudson [8]. As stated before, the resource model will constrain the amount of reevaluation that has to be done.

Note, finally, that this (re)evaluation scheme is still valid even if logical input devices are not used as input stream descriptions. For instance, if the measure and trigger processes of an logical input device were simulated by two basic dialogue cells, (re)evaluation of attributes would still allow the correct resource management.

We first show, by constraining the resource organization in a particular way, how the criteria laid upon the resource manager by the parser can be fulfilled. Second, we show how resource (re)allocation can be done. Algorithms are given that can detect ambiguous situations.

4.1. Resource organization.

4.1.1. Window organization.

Every window belongs to a so-called *window class* which is declared as a set of tiled window descriptions in a virtual coordinate space; i.e.

$$wc = \{w_i \mid \forall\ w_j\ ,\ i \neq j \Rightarrow w_i \bigcap w_j \neq 0\}$$

in which *wc* is the name of the window class and w_i are the names of the window descriptions. Window class declarations must be *instantiated* before a window is used. Instantiation is done by mapping the class declaration onto an already existing window (this window is called the anchor window of the class; thus $w_i \perp wci_{wc_n}$ denotes that the window class instance of window class wc_n is anchored to window w_i). By having a predefined root window class, the collection of instantiations results in a hierarchy in which every node represents an instantiation of a window class.

A *window class instance path* is defined to be the set of window class instances starting form a particular reference window to the root class instance; i.e. † :

$$WindowClassInstancePath_{w_{ref}} = \{wci_n, wci_{n-1}, \cdots, wci_0\}$$

where $w_{ref} \in wci_{wc_n}$ and $\forall wci_{wc_i}\ \exists w_j \mid w_j \in wci_{wc_i} \wedge w_j \perp wci_{wc_{i+1}}$.

Furthermore, we name the set of window class instances that are anchored to a particular window the *AnchoredBy* set of window class instances; i.e.

$$AnchoredBy_{w_{ref}} = \{wci_{wc_i}\}$$

where $\forall wci_{wc_i} \perp w_{ref}$.

The motivation for such a window class organization scheme is three fold :

- By using window classes, a dialogue cell can specify the screen position of its subcells. The resources of these subcells can be specified in terms of the inherited window class.
- Tiled windows within a window class ensures the execution of conceptually different subcells on non-overlapping portions of the screen.
- Anchoring window classes to windows ensures that *all* windows within these classes are related. In particular, moving and resizing an anchor also effects the anchored window class instances.

4.1.2. Logical input device organization.

Similar to windows and window classes, a logical input device belongs to a so-called *input class* which is declared as a set of logical input class type descriptions; i.e.:

$$ic = \{i_j \mid j=1, \cdots, n,\ i_j \in \{L, V, P, C, S\}\}$$

where L, V, P, C,S denote the locator, valuator, pick, choice and string input class types. Instantiation of the input cless is done by associating the class declaration onto an already existing window instance (which, in this case, is also called the anchor and is denoted as $wci_i \perp ici_{ic_n}$).

An *input class instance path* is defined to be the set of input class instances starting from a particular reference window to the root window class instance; i.e.:

$$InputClassInstancePath_{w_{ref}} = \{ici_{ic_i}\}$$

with

$$\forall ici_{ic_i}\ \exists wci_j \in WindowClassInstancePath_{w_{ref}} \mid wci_j \perp ici_{ic_i}$$

† The notation wci_{wc_0} denotes the window class instance of the predefined root window.

The motivation for such an input class organization is two-fold :

- Constraining the set of input devices provides the dialogue designer with a finer control over the distribution of physical input devices during execution.
- Associating input classes with a window ensures that the activation of the input device is restricted to a particular window.

Implementing such a scheme on top of GKS is not possible. A shortcoming of the GKS input model is that the association of measure and trigger processes is fixed by a particular GKS implementation; the application has no control over the way measure and trigger processes are configured to form logical input devices. Additional functionality to the input device model has been defined which allows the run-time system to explicitly configure logical input devices. The key idea is to augment the workstation description table with the number of measure processes of some input class and a number of trigger process. Each measure and trigger process is named and is made visible to the run-time system. Functions are provided for defining a logical input device configuration by specifying which measure and trigger are to used when the device is initialized. Furthermore, inquiry functions allow the details of a particular configuration to be obtained.

The approach of self configurable logical input devices allows the run-time system to maintain its own strategy to associate trigger processes with measure processes. This approach is somewhat similar to the one proposed by Duce [4] as an extension to the GKS logical input device model.

4.2. Resource allocation.

4.2.1. Resource creation.

Resource creation is done by binding each component of the virtual resource description with a particular physical resource. Scope rules are applied to determine which class instantiation is used to select a particular resource component. These rules are determined by examining the resource model in conjunction with the position of the requesting dialogue cell within the derivation tree.

The following scope rules apply to all resource requests:

- The window class instantiation of the requesting window class must belong to the window class instance path of its parent resource; i.e. $wci_{wc} \in WindowClassInstancePath_{r_p[w]}$ with r_p being the parent cell's resource.
- Similarly, the input class instantiation of the requesting input class must belong to the input class instance path of its parent resource; i.e. $ici_{ic} \in InputClassInstancePath_{r_p[w]}$ with r_p being the parent cell's resource.

Scope rules ensure that resource ambiguity resolution can be done efficiently by restricting all other (potential) ambiguous resources to belong in the scope of the resource in question. We now give the ambiguity resolution algorithm in two steps. The first step detects the set of all overlapping windows; i.e. $overlap : window \rightarrow window^{+}$. The second step detects, given a reference resource and the set of resources with overlapping windows, if an ambiguous situation occurs; i.e. $is\_ambiguous : resource \times resource^{+} \rightarrow boolean$

```
01)    $overlap_{\uparrow}(w_{ref})$
02)    [
03)        $S \leftarrow overlap_{\downarrow}(w_{ref}, w_{ref})$
04)        **foreach** $wci_i \in WindowClassInstancePath_{w_{ref}}$
05)        [
06)            **foreach** $w_j \in wci_{i-1}$
07)                **if** $w_j \perp wci_i \neq 0$
08)                    $S \leftarrow S \bigcup overlap_{\downarrow}(w_j, w_{ref})$
09)        ]
10)        **return** $S$
11)    ]

12)    $overlap_{\downarrow}(w_{ref}, w)$
13)    [
14)        $S \leftarrow w_{ref}$
15)        **foreach** $wci_i \in AnchoredBy_{w_{ref}}$
16)        [
17)            **foreach** $w_j \in wci_i$
18)                **if** $w_j \bigcap w \neq 0$
19)                    $S \leftarrow S \bigcup overlap_{\downarrow}(w_j, w)$
20)        ]
21)        **return** $S$
22)    ]
```

Function $overlap_{\uparrow}$ checks, for all window class instances in the the path of the reference window (line 04), if the anchoring window (line 06) contains descendent windows that overlap the reference window (line 07). Function $overlap_{\downarrow}$ recursively checks, for all window class instances that are anchored by a reference window (line 14), if any window in that particular class instance contains an overlapping window (line 16).

Conceptually, $overlap_{\uparrow}$ finds all overlapping windows "higher" than a particular reference window while $overlap_{\downarrow}$ finds all "lower" overlapping windows. The efficiency of this algorithm lies in the fact that

- Only one window per class instance can anchor another class instance (line 06). This is due to the restriction that all windows within the class instance are tiled.
- All descendents of non-overlapping windows do not have to examined (line 16). This is due to the fact that all windows that belong to a particular class instance intersect completely with the anchor; i.e. $w_{descendent} \bigcap w_{\perp} = w_{descendent}$ where $w_{descendent} \in wci \wedge wci \perp w_{\perp}$. Thus, if an anchor does not intersect with a

particular window, all descendent windows of that anchor will not either.

The heart of the function *is_ambiguous* is centered around the function *has_thickness* which determines if a K-intersection exists within a set rectilinear-oriented rectangles. A K-intersection is defined to be point in which at least K rectangles intersect. *Has_thickness* solves the ambiguity problem since, if there exists an $BasicStream_{IsMax} + 1$-intersection, then input tokens from the basic input streams cannot be uniquely distinguished.

The algorithm that implements *has_thickness* is based on the line sweeping paradigm which is very common in the field of computational geometry. The basic idea of line sweeping is to sweep a vertical line from the left of all objects in question to the right of these objects. Each time the sweep line collides with an object an (administrative) calculation can be carried out on the sweep line. In this way a static two-dimensional problem can be reduced into a dynamic one dimensional problem. In this particular case, a so-called *weighted segment tree* is constructed during the sweep. Each external node in the segment tree contains information about the intersection area of a rectangle and the sweep line. A K-intersection can be easily verified by examining the weight factors of the nodes in the tree.

The next section will sketch the K-intersection algorithm applied to rectangles and analyze the resulting order complexities. A generic version of this algorithm which determines is an arbitrary set of polygons has a K-intersection is due to D. Wood [21].

4.2.1.1. Determining a k-intersection within a set of rectangles.

The basic idea is to sweep a vertical line from the leftmost rectangle to the end of the rightmost rectangle. At each position of the sweep line there are rectangles which it intersects (active rectangles), rectangles completely to its left (dead rectangles) and rectangles completely to its right (inactive rectangles). The intersections of each active rectangle with the sweep line results in a number of disjoint intervals (active intervals). Two observations can be made:

- There are at most 2N different sets of active intervals, since the set of active intervals only changes when the sweep line meets a vertical edge of a rectangle. This reduces a continuous sweep of the sweep line to 2n discrete jumps.
- Two rectangles intersect if and only if there is a position of the sweep line for which their corresponding active intervals intersect. This reduces the (two-dimensional) k-intersection problem to a (one-dimensional) sweep-line query problem.

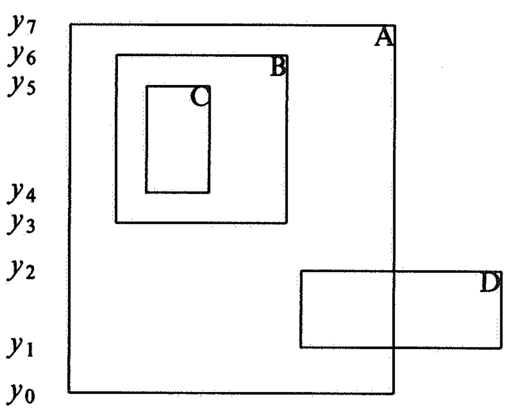
Calling the x-projections of the vertices in ascending sorted order as *sweep points*, we can sketch the intersection algorithm as follows:

```
foreach sweep point x do
  if is_a_left_vertical_edge (x)
  then
    query active intervals with the edge for intersections,
        and insert the edge in the active interval yielding an extended
        active interval or two disjoint active intervals.
    else
        delete the edge from its corresponding active interval
        yielding zero, one or two new active intervals.
```

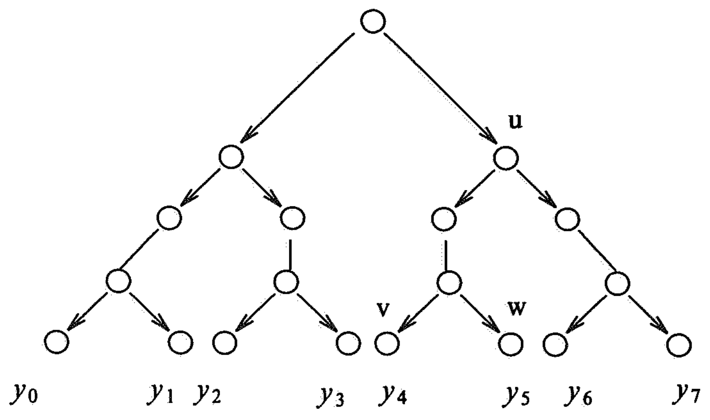
The K-intersection algorithm maintains a *segment tree* which will allow the insert, delete and query operations to be realized in optimal time. A segment tree is a balanced binary search tree with keys as external nodes and internal nodes contain routers which lead the search to a particular key. In this particular tree, the keys are y-values corresponding to the endpoints of vertical edges. Moreover, each external node not only represents a point in the key space, but also the interval in the key space defined by the successor in the tree. This implies that each interval, specified by a pair of keys, is associated with a unique node of the tree. Within the segment tree, an external node y_i represents the interval $[y_i, y_{i+1})$ for $0 \leqslant i < n$ and y_n represents the interval $[y_n, y_n]$. An internal node represents the interval defined by the union of its children intervals.

Example

The following figure depicts four rectangles, resulting in eight vertical edges and giving eight y-values. The corresponding segment tree is given on the right. External nodes are labeled as y_i. The interval of node u in the previous example, denoted as *interval* (u), is the interval $[y_4, y_7)$.



Window layout.



Segment tree.

End Of Example.

Associated with each node u is a set of intervals which cover *interval* (u); i.e.

$$cover\ (u) = \{\ i \mid interval\ (u) \subseteq i \wedge interval\ (\pi u) \not\subseteq i\}$$

where πu is the parent of u. Before the sweep begins *cover* $(u) = \varnothing$, for all nodes u in the tree. In contrast to binary search trees the structure of the tree does not change during insertions and deletions of active intervals. Only the cover sets of the nodes are updated.

For instance, given the empty segment tree in the previous example, inserting the edge $[y_4, y_5]$ of rectangle C causes *cover* $(u) := \{C\}$, *cover* $(v) := \{C\}$, *cover* $(w) := \{C\}$. The latter assignment does not satisfy the rule given above since *interval* $(w) = [y_5, y_6)$. However, because *interval* $(w) \cap [y_5, y_6) \neq \emptyset$ and w is an external node we assign C to *cover* (w) anyway.

Disregarding the technical details of maintaining the active intervals of each rectangle in the segment tree, consider now the K-intersection problem at a particular sweep point. This is equivalent to the query "is there a position on the sweep line with k intersecting active intervals ? " By maintaining the cardinality of each cover set in an extra field (denoted as $\#cover\ (u)$), the algorithm to determine the thickness is now straightforward. The thickness of each node is given as

$$thickness\ (u) = \max\ (thickness\ (\lambda u),\ thickness\ (\rho u)) + \#cover\ (u)$$

where λu (respectively ρu) denote the left (respectively right) child of node u. The thickness is synthesized throughout the segment tree after each insertion and deletion.

If a segment tree has n external nodes then its height is bounded by $\lceil \log n \rceil$. Insertion only affects the cover sets of at most 4 $\lceil \log n \rceil$ nodes. Deletion is somewhat more tricky, but can still be carried out by maintaining an additional table to pointers. Hence, the complete K-intersection algorithm can be realized in $O\ (n \log n)$ time.

4.2.2. Resource redistribution.

Resource redistribution is characterized by the amount of flexibility the user has in dynamically changing resource descriptions. We consider only two cases of resource redistribution: 1) a simple form of window redistribution in which the user can change the window layout and 2) input device redistribution in which the user can change the mapping between logical and physical devices.

Resource redistribution is done in synchronous mode, ensuring that all other operations using resources are temporally disabled. In all cases, however, resource redistribution must result in a non-ambiguous resource model so that all operations are able to continue without having the knowledge that the resource model has changed.

4.2.2.1. Window redistribution.

In this section we only consider the moving of a window and the impact of this on the resource model and the segment tree. Other window operations, such as resizing , exposing, hiding and closing can be realized similarly.

As was shown in a previous section, the resource model defines a hierarchy of windows. Due to the many-to-one relation between the resource hierarchy and the dialogue cell hierarchy, moving a particular window implies moving all descendent windows belonging to the resources of descendent cells. This is desirable since the resources of each descendent dialogue cell are defined in terms of their parent resources. However, in order to be able to keep the resource model intact, two restrictions are laid upon the move operation:

- The window in question must stay within the bounds of the anchoring window. Restricting window movement in this way is not as bad as one suspects. The rationale behind this is that since a (grand)parent dialogue cell is active in the anchoring window, the pictorial output of the window in question is related to that of the anchoring window.
- The intersection of neighboring windows within the window class instance must stay empty. One of the main reasons for specifying two distinct windows in one class is to ensure that unrelated dialogue cells are active in disjunct screen spaces. By restricting the intersection of unrelated windows to be empty, this relationship is maintained.

After every move operation the segment tree is reconstructed. Fortunately, the structure of the segment tree does not change. Only the y-values of those external nodes which lie within the interval of the anchor window obtains new values, and hence, only the cover sets of that portion of the tree has to be reevaluated.

For instance in the example given in section 4.2.11, moving window B is restricted to the interval $[y_0,y_7)$. In this particular example, moving window B implicitly means that window C be also be moved. In this case, the external nodes with values y_3,y_4,y_5,y_6 obtain new values.

Since only a sweep from x_0 to x_n is needed, reconstructing the segment tree and determining if the move results in an ambiguous situation can be done in O (n log n)-time where n is the number of windows that intersect the anchoring window.

4.2.2.2. Input device redistribution.

A second form of resource redistribution allows a user to dynamically change the the mapping from a logical to physical input device. Although the resource model is expressed in terms of logical input devices, somewhere the mapping onto physical input devices takes place. Since this mapping is done *outside* the resource model it is an implementation issue how it is realized. However, the functionality of the resource manager is such that the user is allowed to overrule this mapping.

For instance, assume that a logical locator device and a logical pick device are mapped onto two different physical buttons. The user must be able to dynamically change this mapping by indicating another button configuration.

It is the resource manager's responsibility to ensure that ambiguous situations do not occur. This can imply that other mappings will have to be changed as well. However, since an unambiguous situation is possible before the input device redistribution, an unambiguous configuration also exists thereafter.

5. Conclusions.

In this paper we have discused the role of a resource management scheme within a dialogue language. Using resources as the basis for input and output results in the coupling of input and output at the physical device level. By integrating the resource manager in the dialogue language, correlations of higher level input and output can be defined and maintained.

Scope and ambiguity rules are defined for resources as natural extensions to the corresponding activation rules of dialogue cells themselves. Moreover, by relying on the concept of inheritance, it has been shown how partial resource specifications are elaborated.

From the resource specifications, a resource model is constructed. This model can also be dynamically manipulated by the user. Changes at one level of the resource model are implicitly propagated towards lower levels. However, the resource manager must ensure that the constraints laid upon the model by the dialogue specification are satisfied.

Parallelism is a key issue in dialogue cell languages. Since multiple transactions can take place within one window, ambiguous situations can occur when an input device is triggered. An $O(n \log n)$-time algorithm is given which detects if a resource request results in such a situation.

Future work will include the generalization of the resource concept to include for instance, CPU time and memory capacity. Moreover, the resource model will be expanded to include *all* type and object descriptions within the dialogue language. Here too, the user must be able to dynamically change the type and object specifications whenever possible. This is the first step to truly interactive systems in which the user is in command of the program.

At the implementation level, work still has to be done on mapping the resource manager onto arbitrary "off-the-shelf" window managing packages and comparing the resulting efficiency of these mappings. Currently, the resource manager is implemented on top of a local version of GKS which includes some primitive support for window management [16].

Appendix 1.
Constructing an equivalent context-free grammar from a dialogue grammar.

Here we show how an equivalent context-free grammar can be constructed from a dialogue grammar. The two grammars are defined to be equivalent if the corresponding languages are equivalent.

Suppose a dialogue grammar is given by :

$$G_A = <\Sigma_{N_A}, \Sigma_{T_A}, P_A, Z_A>$$

The derivation algorithm applies one of the following primitive cases :

1. Assume that P_A contains the production rule $X \rightarrow ; (E_1, \cdots, E_n)$. Each E_i is a well-formed symbol subexpression. The symbol expression X can be be rewritten into the set of $n + 1$ rules:

   $$\{X \rightarrow H_1 \cdots H_n, H_1 \rightarrow E_1, \cdots, H_n \rightarrow E_n\}$$

2. Assume that P_A contains the production rule $X \rightarrow \vee (E_1, \cdots, E_n)$. The symbol expression X can be be rewritten into the set of $n + 1$ rules:

   $$\{X \rightarrow H_1, H_1 \rightarrow E_1, \cdots, H_1 \rightarrow E_n\}$$

3. Assume that G_A contains the production rule $X \rightarrow \wedge (E_1, \cdots, E_n)$. The symbol expression X can be be rewritten into the set of $1 + n$ rules:

   $$\{X \rightarrow X_1, \sum_{i=1}^{n} X_1 \rightarrow \prod_{i=1}^{n} (H_{i_1} \cdots H_{i_n}), H_1 \rightarrow E_1, \cdots, H_n \rightarrow E_n\}$$

   In this case, $\prod_{i=1}^{n} (H_{1_i} \cdots H_{1_i})$ indicates a permutation over the symbols H_i , $i = 1 \cdots n$. The expression

   $$\sum_{i=1}^{n} X_1 \rightarrow \prod_{i=1}^{n} (H_{i_1} \cdots H_{i_n})$$

   indicates the set of production rules having Z_1 as left hand side and a permutation of X_{i_j} as a right hand side.

4. Assume that P_A contains the production rule $X \rightarrow * (E)$. The symbol expression X can be be rewritten into the set of 3 rules:

   $$\{X \rightarrow \epsilon, X \rightarrow X H_0, X_0 \rightarrow E\}$$

5. Assume that P_A contains the production rule $X \rightarrow + (E)$. The symbol expression X can be be rewritten into the set of 3 rules:

   $$\{X \rightarrow X H_0, H_0 \rightarrow E\}$$

6. Assume that G_A contains the production rule $X \rightarrow$ **case** $(E_1, \cdots, E_n)$. The symbol expression X can be be rewritten into the set of n rules:

   $$\{X \rightarrow H_2, H_2 \rightarrow E_1 E_2, H_2 \rightarrow E_1 E_3, \cdots, H_2 \rightarrow E_1 E_n\}$$

Intuitively, is derivation algorithm recursively applies the six different cases to the root production rule Z_A until only symbols in set $\Sigma_{N_A} \bigcup \Sigma_{T_I}$ are on the right hand side of the generated production rules. This process eventually results in the equivalent context-free grammar, denoted as

$$G_A^{CF} = <\Sigma_{N_A}^{CF}, \Sigma_{T_A}^{CF}, P_A^{CF}, Z_A^{CF}>$$

Assuming that the root symbol expression of the original dialogue grammar (without combining operators) has the form $A \rightarrow A_1 \cdots A_n$, then production rules of G_A^{CF} are the set:

$$P_A^{CF} = P_{A_1}^{CF} \bigcup \cdots \bigcup P_{A_n}^{CF} \bigcup \{ \cdots \}$$

Here $P_{A_i}^{CF}$ are the productions of the grammar which was derived from the dialogue grammar G_{A_i} and $\{ \cdots \}$ indicates the set of productions which are derived by recursively applying the six cases to the root symbol expression. Furthermore, $\Sigma_{N_A}^{CF} = \Sigma_{N_A} \bigcup \{H_{ij}\}$ and $\Sigma_{T_A}^{CF} = \Sigma_{T_A}$. H_{ij} are the extra non-terminal help symbols that were introduced while deriving the context-free productions.

Semantic actions are associated to the corresponding help non-terminals H_{ij}. These are evaluated whenever the parser reduces the corresponding non-terminal. Whenever the context-free production rule corresponding to the dialogue symbol expression Z_A^{CF} is reduced, a semantic action produces a value. This value can be used in the semantic actions of the rules which use Z_A^{CF}. In the special case that the production rules derived from derivation rule 4, 5 or 6, the value determines if the parser is to make a shift or a reduce decision.

A second aspect of the derivation algorithm concerns the activation mode in which a dialogue cell was activated. A dialogue cell activated in an asynchronous mode (viz. sample or event) can produce can produce a result a number of times before it is deactivated. This can be expressed in a context-free grammar as follows: assume that the dialogue cell , A, is activated in asynchronous mode and that the dialogue grammar is denoted by

$$G_A = <\Sigma_{N_A}, \Sigma_{T_A}, P_A, Z_A>$$

The equivalent context-free grammar is written as (in this case an extra subscript is used to indicate that A is activated in asynchronous mode) :

$$G_{A_a}^{CF} = <\Sigma_{N_{A_a}}^{CF}, \Sigma_{T_{A_a}}^{CF}, P_{A_a}^{CF}, Z_{A_a}^{CF}>$$

in which

$$P_{A_a}^{CF} = P_A^{CF} \bigcup \{Z_{A_a}^{CF} \rightarrow H_0 \text{ , } H_0 \rightarrow Z_A^{CF} H^0 \text{ , } H_0 \rightarrow Z_A^{CF}\}$$

and $\Sigma_{N_{A_a}}^{CF} = \Sigma_{N_A}^{CF} \bigcup \{Z_{A_a}^{CF} + H_0\}$.

Appendix 2.
Attributed dialogue grammars.

Attribute grammars ([9] , [10]) allow the semantics of a language to be specified along with its syntax. Underlying an *attributed dialogue grammar* is its dialogue grammar

$$G = <\Sigma_N, \Sigma_T, P, S>$$

where Σ_N and Σ_T are the disjoint finite sets of *dialogue cell symbols* (= non-terminals) and *basic dialogue cell symbols* (= terminal), respectively; P is the set of symbol expressions and $S \in \Sigma_N$ is the root cell symbol.

With each cell symbol we associate two finite disjoint sets; a set of *inherited attributes* and a set of *synthesized attributes.* Inherited attributes obtain values from the immediate parent node *and* its production in the derivation tree. Synthesized attribute values are computed from the attribute values of the immediate descendents in the derivation tree. Rules are associated with the context free productions for the evaluation of the attribute values.

The inherited attributes on the left hand side and the synthesized attributes on the elements of the right hand side represent values obtained from the surrounding nodes in the derivation tree. In this sense, inherited attributes represent information that is passed down from the root node towards the leaves. Synthesized attributes of a node represent information which is derived in the subtree of the node and passed up towards the root node of the derivation tree.

With each production $p : A_0 \rightarrow A_1 , \cdots , A_k$ we associate a set of semantic functions. Each semantic function defines a value for a synthesized attribute of A_0 or an inherited attribute of $A_i ,i \leqslant k$. The functions can be defined in terms of (other) attributes of $A_0 , A_1 , \cdots , A_k$.

A derivation tree node, labeled A, defines a set of *attribute instances* corresponding to the attributes of A. A *semantic tree* is a derivation tree together with an assignment of either a value or the special token ω to each attribute instance of the tree. To determine the "meaning" of a dialogue, one first constructs its semantic tree with an assignment of ω to each attribute instance, and then evaluates the semantic functions of as many attribute instances as possible. The latter process is termed *attribute evaluation.* The order in which attributes are evaluated is arbitrary, but is subject to the constraint that a particular semantic function can be evaluated only when all of the argument attributes are available; i.e. non-ω. A semantic tree is fully attributed if a value is associated with each of its attribute instances; it is partially evaluated if the value of at least one attribute instance is unavailable. An attribute grammar is well-formed if every complete derivation tree can be fully attributed.

Knuth stated ([9]) the conditions that attributes must obey in order to evaluate these semantic functions. Relationships among attributes in a given derivation tree are represented by a *dependency graph.* Nodes in the dependency graph represent an attribute instance and there is a directed edge from node a to b if the semantic function for the attribute corresponding to instance b has attribute a as an argument. The basic idea of Knuth is to show that the dependency graph is acyclic if and only if the attribute grammar is well-formed.

The concept of attribute grammars is not a complete method for making formal definitions. For general use, it must be combined with a method for the specification of its evaluation rules. A more formal approach to specifying semantics of context free grammars can be found in [11].

In this paper we will consider only attributes that influence the management of resources.

Appendix 3.
Logical input devices.

The GKS input model is based on the concept of logical input devices. Logical input devices provide the application program with an interface which abstracts physical input devices from a particular hardware configuration.

A logical input device consists of:

- a *class*.
  The class of a logical input device defines the type of the input value which is returned. The six different classes are given in the following table :

device	returntype
locator	Wc, Ntran
choice	Choice
pick	Pickid
valuator	Value
string	String
stroke	Wc [1, · · · , n], Ntran

The GKS logical input classes.

  The actual amount of logical devices belong to each class is workstation dependent. Each individual logical input device within a class is distinguished by a unique number.

- a *mode*.
  The activation mode indicates how the input value is obtained from the logical input device. Conceptually, there are always two processes running for each active logical input device; these are the so-called *measure process* and *trigger process*. A particular measure value of a logical input device is defined to be the (eventually transformed to device coordinates) value of the physical input device.
  The measure process always contains the current measure value of the logical input device. Usually, the measure value is echoed in some way on the screen, for instance, by echoing a cursor shape in the position that corresponds with the measure value.
  A trigger process is an independent, active process that, when *triggered* by the user, sends a message to the measure process. Triggering a logical input device indicates that the current measure value must be returned to the application.

  How the measure value is mapped onto a value returned by a logical input device is defined differently for every input class. For the *locator* device the mapping rules are:

  ⊕ Transform the measure value (given in device coordinates) back to normalized device coordinates using the inverse of the current workstation transformation.

- ⊕ Select the normalization transformation with the highest viewport input priority in whose viewport the normalized coordinate lies. The selection of a normalization transformation always succeeds since there is a default normalization transformation which covers the complete normalized device coordinate space.
- ⊕ transform the normalized coordinate back to a world coordinate using the inverse of the selected normalized transformation.
- ⊕ return the world coordinate and the selected normalized transformation to the application program.

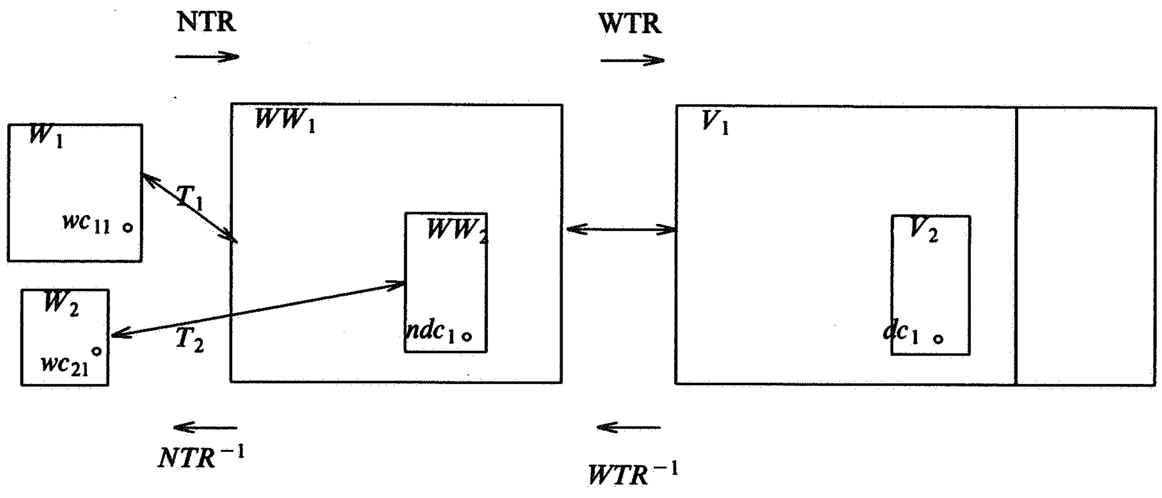
There are three different activation modes:

- ⊕ *request*
  In the case of request mode, the application program waits until the trigger process sends a message to the measure process. The value of the measure process at the moment of triggering is (after the necessary transformations) passed to the application program.
- ⊕ *sample*
  In the case of sample mode, the value of the measure process will, at the moment of sampling, be passed to the application program. No triggering is involved when a logical device is sampled so that the application program will immediately continue after issuing a sample call.
- ⊕ *event.*
  In the case of event mode, the application program does not wait until the trigger process sends a message to the measure process. However, when the logical input device is triggered the value of the measure process at the moment of triggering is put in a input queue. The contents of the queue can be acquired by the application program by issuing calls that query and get the queue elements.

- *attributes.*
  Attributes are used to parameterize the initialization of a logical input device. Most attributes have to do with how and where input devices produce echos on the screen. Attributes include initial values, prompt / echo types, activation modes and echo areas. Data records provide the application program a means to parameterize the logical input device in a device dependent manner. The layout of a data record must be precisely specified in the installation guide of a particular implementation. For instance, an entry in the data record can specify which mouse button will be used to trigger a locator device.

Example

This example illustrates how a value from *locator* input device is transformed back to world coordinates. Suppose an application program has defined two window / viewport transformations (T_i from W_i to WW_i) and uses the default workstation transformation at the moment the *locator* device is triggered. The following figure illustrates the different coordinate spaces and the relevant transformations. †



Input transformations.

If the physical locator device triggers at the point dc_1 then, in accordance with rule 1, the inverse of the current workstation transformation is used to calculate point ndc_1. There are now two cases which must be distinguished:

1. viewport input priority $(T_1) >$ viewport input priority (T_2)
   Rule 2 selects T_1. The inverse of this transformation calculates the point wc_{11}. Finally, wc_{11} and T_1 are returned to the application.
2. viewport input priority $(T_1) <$ viewport input priority (T_2)
   Rule 2 selects T_2. The inverse of this transformation calculates the point wc_{21}. Finally, wc_{21} and T_2 are returned to the application.

End of Example

† For simplicity reasons, the default window / viewport transformation is not shown.

References.

1. H. G. BORUFKA, H. W. KUHLMANN and P. J. W. HAGEN, Dialogue Cells: A Method for Defining Interactions,, *IEEE Computer Graphics and Applications*, **2**(6), July 1982., pp. 25-37.
2. K. DE VLAMINK and L. LEWI, *A programming methodology in compiler construction*, North-Holland, Amsterdam, (1982).
3. A. DEMERS, T. REPS and T. TEITELBAUM, Incremental context-dependent analysis for language based editors, *Trans. Prog. Lang and Systems*, **5** , 1983, pp. 449-477.
4. D. DUCE, *Configurable input devices - a discussion paper*, Rutherford appleton laboratory, Chilton, (1987).
5. A. GOLDBERG and D. ROBSON, *Smalltalk-80 the language and its implementation*, Addison Wesley, Reading, MA., (1983).
6. M. W. GREEN, *The design of graphical user interfaces*, University of Toronto (CSRI-170), Toronto, (1985).
7. R. C. HOLT, J. R. CORDY and D. B. WORTMAN, Introduction to the syntax / semantics language, *Trans. Prog. Lang and Systems*, **4**(2), 1982, pp. 149-172.
8. S. E. HUDSON and R. KING, The Higgens UIMS and its efficient implementation of undo, in *Foundations of computer sceince.*, J. EARNSHAW (ed.), (1986).
9. D. E. KNUTH, Semantics of context-free languages, *Mathematical systems theory journal*, **2** , 1968, pp. 127-145.
10. D. E. KNUTH, Semantics of context-free languages: correction, *Mathematical systems theory journal*, **5** , 1971, pp. 95-96.
11. M. MARCOTTY, H. F. LEDGARD and G. V. BOCHMAN, A sampler of formal definitions, *Computer surveys*, **8**(2), 1968, pp. 191-276.
12. W. M. NEWMAN, A system for interactive graphical programming, *J. Computer and System Sciences*, (1968), pp. 38.
13. D. OLSEN and E. DEMPSEY, SYNGRAPH: a graphical user interface generator, *SIGGRAPH'83*, (1983), pp. 43.
14. H. J. SCHOUTEN and P. J. W. TEN HAGEN, *Parallel graphical output from dialogue cells*, CWI report (CS-R8719), Amsterdam, (1987).
15. P. J. W. TEN HAGEN and J. DERKSEN, *Parallel input and feedback in dialogue cells*, CWI report (CS-R8413), Amsterdam, (1985).
16. P. J. W. TEN HAGEN and M. M. DE RUITER, *Segment grouping, an extension to GKS*, CWI report (CS-R8623), Amsterdam, (1986).
17. R. VAN LIERE, *Scheduling in DICE* , CWI IS memorandum, Amsterdam, (1986).
18. R. VAN LIERE and P. J. W. TEN HAGEN, *A model for graphical interaction*, CWI report (CS-R8718), Amsterdam, (1987).
19. R. VAN LIERE and P. J. W. TEN HAGEN, *Introduction to dialogue cells*, CWI report (CS-R8703), Amsterdam, (1987).

20. J. VAN DEN BOS, M. J. PLASMEIJER and P. H. HARTEL, Input-Output tools: a language facility for interactive and real-time systems, *IEEE Transactions on Software Engineering*, **9** , 1983, pp. 247-259.

21. D. WOOD, An isothetic view of computational geometry, in *Computational geometry*, G. T. TOUSSAINT (ed.), North-Holland, Amsterdam, (1985).