LA-UR 9.7 - 5032
9 December 1997

RECEIVED

APR 06 1998

OSTI

ExtendTM Customization - Experiences and Issues

Robert Y. Parker

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

DTIC QUALITY INSPECTED 4

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

*Los Alamos National Laboratory is operated by the University of California for the United States
Department of Energy under contract W - 7405 - ENG - 36*

19980423 148

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Table of Contents

1.0 Introduction	3
2.0 Extend™ Background	3
2.1 Extend Blocks	3
2.2 The Underlying Structure of Blocks	4
2.3 Continuous and Discrete Event Modeling in Extend™	4
2.4 Approaches to Customizing Extend™	5
3.0 Extend™ Block Customization Efforts	6
3.1 Scope of Customization Efforts	6
3.2 Why Customize?	6
3.3 Challenges, Disadvantages of Customization	7
3.4 Advantages of Customization	8
3.5 Specific Custom Block Work	9
3.6 Impact of Customization: Specific Examples	9
4.0 Conclusion	10
References	11
Appendix	12

Extend and ModL are trademarks of Imagine That Inc. Macintosh is a registered trademark of Apple Computer, Inc. Windows is a trademark of Microsoft Corporation. All other product names used herein are trademarks of their respective owners.

1.0 Introduction

Extend™ simulation software is a dynamic modeling package developed by Imagine That Incorporated. The Technology Modeling and Analysis group (TSA-7) at Los Alamos National Laboratory has used Extend extensively over the past few years as one of various tools employed to perform simulation modeling and analysis. Development efforts over much of this period have made Extend a more effective and efficient tool through block customization. TSA-7 has taken advantage of the built-in capability in Extend to allow users to create new or modify existing functional blocks from which simulation models are constructed. As a result, Extend is much more effective and efficient for the group's applications.

This paper summarizes block customization and simulation model development that markedly improved the utilization of the Extend software package. The basis for the paper is work conducted intermittently between September 1995 and October 1997. The material covered herein includes some background information on Extend, which is necessary for understanding the balance of the paper. Following the background, the paper addresses Extend block customization efforts, including advantages and disadvantages to customizing, and the impact customization has had on Extend modeling efforts in TSA-7. Brief descriptions of many customized blocks developed by the author are presented in the appendix.

2.0 Extend Background

Extend is a dynamic modeling tool that is designed to run under Mac OS and Windows operating environments. Whereas static modeling is usually used to represent a snapshot in time of the state of a system, dynamic modeling (as in Extend) helps capture the time-based behavior and inter-relationships of components of the system modeled. Extend can dynamically model systems in either a continuous or discrete event fashion. Extend offers the power to quickly and conveniently build simulation models graphically, as well as to customize and expand the capabilities of such models through an underlying programming language. As such, Extend is both a graphic simulator and a simulation language hybrid. These characteristics and additional background information are addressed in the following sections.

2.1 Extend Blocks

Extend has a convenient graphical user interface with libraries of general-use functional blocks that allow users to quickly build models. Each block in the libraries performs certain functions. To use a block one "drags" a copy of the block from the library list of blocks onto the workspace. Each block has an associated dialog box that appears when the block is opened. Through the dialog box, parameters and other information relating to the block are entered and displayed. Most standard Extend blocks connect to each other via connection lines that transmit information between icon connectors. Several functional blocks connected together can represent a complex

process. This is the primary way Extend models are built, connecting different functional blocks together to represent the desired behavior. Such a procedure makes Extend an easy-to-use graphical simulator.

Extend models can be organized hierarchically. That is, groups of interconnected blocks can be grouped into a single hierarchical block (H-block). The H-block is a single icon that when opened reveals the set of blocks nested inside of it. This is extremely useful for organizing a model into different levels for easier understanding and better visual presentation. The nature of H-blocks also makes capturing and reusing the combined function of many interconnected blocks very easy. For example, several standard blocks may be configured to represent a complex workstation with certain behaviors. This set of blocks can be saved into a single H-block. That single H-block can then be connected to other blocks and used in the model just like a standard block is used. The H-block can also be copied to create multiple instances of the workstation construct. Furthermore, the H-block workstation can be saved to a library and used to build other models.

2.2 The Underlying Structure of Blocks

Each standard Extend block is a pre-programmed object that performs a certain function. It is "pre-programmed" in that the underlying structure of the block was already created to perform the desired function. While certain aspects of the underlying modeling structure are not available to the user, the primary structure through which non-hierarchical-type blocks are created is accessible to the user. Hence, users can create their own Extend blocks or modify existing ones. The underlying components of the structure of Extend blocks include the block's dialog, icon, help text, and script. The script is the written code that dictates block operations. In Extend, the script is coded in a programming language called ModL, which is characterized as a subset of the C programming language. The programmable ModL code gives Extend the flexibility of a simulation language.

2.3 Continuous and Discrete Event Modeling in Extend

Simulation tools are typically classified as continuous or discrete event. Continuous simulation tools are often used to model systems whose state changes continuously over time, such as the volume of a tank at a chemical processing facility. Values and calculations in continuous models are generally updated at evenly spaced time intervals. Discrete event simulation tools, on the other hand, are often used to model systems whose state changes as a result of certain events occurring at discrete points in time, such as a manual assembly line in a manufacturing plant. Hence, the intervals at which calculations are updated in a discrete event model depend on when the events, which are rarely evenly spaced, occur. Extend can model both continuous and discrete changes in systems.

The characteristics of Extend continuous simulations are captured by values that are passed between the different blocks in the model and recalculated at every time step.

A continuous simulation typically only provides general statistics about the system being modeled. Information about individual entities in the system is not tracked because flows, not individual entities, are modeled. The underlying ModL code in Extend blocks used for continuous modeling is generally straightforward and simple. The primary means of transferring information between blocks is by checking the values at connectors. Additionally, checking values and recalculating equations usually occurs at the same time step and at equally spaced intervals for every block in the model. Hence, there is no need to handle time in a special way, nor to track much more information than the values being generated by the blocks.

The characteristics of Extend discrete event simulations are captured not only by values that are passed between blocks in the model, but also by attributes and priorities that are assigned to individual entities (items) in the model. Discrete event simulations typically provide information about each individual item in the model in addition to general system statistics. In contrast to continuous models, the state of a discrete event model is not directly linked to the mere passing of time, but to the occurrence of specific events at discrete points in time that are typically not evenly spaced. Some locations in the model may perform operations (calculations, etc.) at some event times but not at others. Because of the more detailed information tracked by discrete event models and the uneven time steps, Extend blocks used in discrete event models have more complex underlying code than continuous blocks to accommodate the detailed item information and the discrete event timing. Since Extend discrete event blocks act based on system events occurring around them, they require a network of message passing to communicate with the blocks around them. This message passing allows the blocks to assess the state of the model around them so they can act only as needed. The message passing contributes significantly to the underlying code required for the blocks.

2.4 Approaches to Customizing Extend

Blocks in the standard Extend libraries are, for the most part, designed to be used together. They follow the same underlying paradigm and are made to work under the same type of global architecture. This allows almost any standard Extend block to work with almost any other standard Extend block. When customizing Extend by creating or modifying blocks through the underlying ModL code, the issue of local versus global customization must be addressed.

Local customization can be defined as customizing at the functional block level without requiring a change in the way the block interacts with other standard Extend blocks. This means that a locally customized block can still be used in any model with blocks from the standard Extend libraries. The custom block follows the same rules and architecture as standard Extend blocks do. Local customization is good for creating general-use functionality that can be used in any model under the standard Extend modeling paradigm.

Global customization can be defined as both customizing at the functional block level and changing the way the blocks interact with each other. This means that globally

customized blocks follow an architecture or paradigm that may not be the same as is followed by the blocks in the standard Extend libraries. Globally customized blocks are not necessarily intended to work with other standard Extend blocks, but with other globally customized blocks that follow the same set of rules. As a consequence, models developed using a global customization scheme are likely to store or exchange information in a different way than standard Extend models do. Global customization is good for creating models that manipulate or deal with data and information in specialized ways (for example, special global data structures). Globally customized blocks are not as "portable" as locally customized blocks, but they often achieve specific functionality more efficiently.

Extend customization does not need to be strictly local or global. It can include a combination of both. For example, a block can be locally customized to work under the standard Extend paradigm, but require global control from other custom blocks to perform a specific function. In this case, the degree to which the block is globally customized is probably low, but it still has elements of both approaches.

3.0 Extend Block Customization Efforts

The built-in capability in Extend to modify existing and create new functional blocks through ModL scripting is denoted *block customization*. The following several sections address various points and issues experienced by the author with block customization.

3.1 Scope of Customization Efforts

Much of the customization work to date has been the development of many locally customized, general-use and specific-use blocks employed in a variety of modeling and analysis efforts. These blocks were constructed and coded consistent with the structure of existing standard Extend blocks and are intended to interact with other blocks in the standard Extend manner. In the remainder of the paper, any description of customization efforts or further mention of customization refers to and is based upon this local customization work.

3.2 Why Customize?

Many basic functions can be achieved with individual standard Extend blocks. Added functionality can be gained by connecting many blocks together in certain configurations to obtain a desired behavior not otherwise available from any single standard block. Such a configuration can even be made into a hierarchical block and saved in a library so that the particular behavior of the group of blocks can be easily accessed and used repeatedly in a model. Interconnecting several blocks in clever ways allows more complex behaviors to be captured without requiring knowledge of programming or the underlying structure of the blocks.

An example is the modeling of a generic manufacturing workstation that processes a part and has various output paths (product, waste, rework, etc.). When attributes on items are used to represent material quantities and the workstation must perform a "material-balance," then the workstation may require 20 or 30 standard blocks together to properly account for and divert materials to the proper output paths. While this technique of building higher-order capability assists greatly in rapid model development, using so many blocks together to represent a single workstation increases the size of the model significantly as workstations are added. A model of a system with many workstations would quickly become very large and cumbersome; other disadvantages include longer times required to load and run the model, more difficulty debugging the model, and large file sizes.

The disadvantages of the previous example help illustrate some of the benefits of customizing. The idea is to create a single custom block through ModL programming that would achieve the same behavior as most or all of the combined 20 or 30 blocks in the workstation configuration. Hence, customization can (1) simplify models, (2) decrease the size of models, (3) decrease the time to load models, (4) increase the speed of models, and (5) achieve behaviors and characteristics not otherwise possible (or very difficult) to achieve using interconnected standard blocks.

3.3 Challenges, Disadvantages of Customization

Several challenges and disadvantages present themselves in the customization of Extend blocks. Customization requires ModL programming. The extent of the programming or code modification depends on the extent and complexity of the customization desired. Learning ModL scripting can present a challenge, especially to a user with no previous programming experience. Since the ModL language is very similar to ANSI C standard, a user with experience in C or some other programming language will likely have an easier time learning ModL scripting techniques than someone who has no programming experience. Although relatively simple customization can be mastered in a few days, a few months of extensive practice may be needed to become proficient at more complex customization tasks.

The process behind discrete event modeling in Extend is quite a bit more complex than continuous modeling with Extend. Continuous modeling involves blocks that primarily involve passing and manipulating values between blocks at continuous time steps. The ModL script behind standard blocks used in continuous modeling (like those from the Generic Library) is simpler than that of blocks used in discrete event modeling (like those from the Discrete Event Library). Hence, the creation of custom blocks used for continuous modeling is much simpler than for discrete event modeling. The challenge of discrete event blocks comes from their need to deal with many more types of information such as values, attributes, variables of several types, discrete time steps, etc. The complex system of message passing that occurs between the blocks of a discrete event model is more difficult to understand and account for in block scripts. This challenge can be more easily addressed by creating custom blocks based on the code in existing standard blocks. In fact, the Extend manual encourages users to start with

standard blocks and modify them to make custom blocks. In this way, the essential parts of code relating to discrete event modeling are easier to integrate and incorporated, although care must still be taken to integrate and incorporate these message passing and message handling pieces of code correctly. For instance, in one particular case a potential problem was discovered in conjunction with a certain configuration of custom and standard blocks in a model. The problem was attributed to excess messages being generated and passed in the model in such a way that the messages “snowballed” and slowed down the execution. When a temporary fix to the problem was achieved, model execution speed increased 12 to 15 times.

Imagine That Incorporated offers good customer support for users of Extend, and they constantly maintain the blocks in the standard libraries provided with the software. They deal with any bugs or problems found in the standard blocks, but are not necessarily responsible for problems or bugs in custom blocks. Thus, the creator of custom blocks must try to ensure that the blocks function properly and assume responsibility for the maintenance of the custom blocks. If long-term use of custom blocks is anticipated, someone must be available to maintain and modify the blocks.

3.4 Advantages of Customization

There are several advantages to customizing Extend blocks. Customizing often helps simplify and improve the efficiency of complex models. If recurring instances of a particular configuration of interconnected blocks are present in a model and are likely to be used in other models, the configuration may be a good candidate for customization. Replacing the entire configuration with a single custom block reduces the number of blocks, leads to a less cumbersome model, and reduces the file size. Furthermore, the time to load the model is reduced. Since a lot of message passing occurs between blocks in discrete event models, the number of messages being passed can be reduced significantly by replacing a group of blocks with a single custom block. Eliminating the extra blocks also reduces the total number of procedure and function calls in the underlying code, and model execution speed increases. The speed and size improvement of a custom block over the corresponding set it replaces depends on the number and types of blocks actually replaced. A single custom block in the mix with a number of other blocks in a large model will not always result in obvious improvements. However, if the single block is used many times or if different custom blocks are used throughout the model as well, then significant speed and size improvements can be achieved.

Customizing often enhances the functionality and flexibility of models and blocks. Frequently a particular behavior or characteristic that seems difficult to achieve using standard blocks can be more easily achieved by creating a custom block. Work-arounds for problems or tricky logic using standard blocks are frequently necessary when working with complex models. These work-arounds can sometimes be unreliable, may not address the problem in exactly the right way, and may limit the flexibility of the simulation. Customizing generally results in a more robust solution and often allows for enhanced functionality to be incorporated. Customizing also allows for more error

checking and crash-proofing to be built in, frequently making the model more flexible and user-friendly.

Finally, learning how to customize blocks provides a better understanding of the structure of Extend. Familiarity with the underlying structure of the blocks and comprehension of the detailed workings of the code in those blocks leads to a better understanding of the behaviors and nuances of Extend. Understanding the code in the blocks is helpful in debugging models and is useful in clarifying the behavior of the blocks.

3.5 Specific Custom Block Work

Over the past two years extensive experience in local block customization has been gained. Many of the custom blocks have replaced all or portions of existing hierarchical blocks. These H-blocks performed recurring functions in many models built by other members of the group. More custom blocks were created as certain modeling structures became well defined and frequently used. Approximately 85% of the custom blocks created by the author were discrete event blocks, since most of the modeling used this capability. The majority of these custom blocks were organized into two separate libraries, the *RYP Lib1* library and the *RYP(A-D) Lib* library, which were released for general use to Extend users in the group and outside associates and customers. As of December 9, 1997 the *RYP(A-D) Lib* library contained 26 blocks and the *RYP Lib1* library contained 51 blocks. The blocks in the *RYP(A-D) Lib* library are actually standard Extend blocks that have been modified to display more information in the blocks' dialog boxes. Most of the blocks in the *RYP Lib1* library required more extensive customization and continue to be maintained and updated. A chart showing the 51 blocks of the *RYP Lib1* library, along with a brief description of the function of each block, is included in the Appendix.

In addition to the general use blocks provided in the *RYP Lib1* and the *RYP(A-D) Lib* libraries, many custom blocks were created in conjunction with specific models and projects. While these project-specific blocks can still be used in models with standard Extend blocks, they are not likely to be used in other models since their functions are more applicable to specific project needs. The functionality provided by these blocks in specific models was necessary and frequently required less work to develop than devising an equivalent configuration of standard blocks to perform the same task.

3.6 Impact of Custom Blocks: Specific Examples

The positive impact that block customization can have on Extend modeling efforts has already been addressed generally. At this point a few specific examples will be given to better illustrate the benefits of customization.

The "Separator (Simple)" block from the *RYP Lib1* library is a custom discrete event block which separates a single input item into multiple quantities of output items. The attributes on the incoming item can also be separated and attached to the various

output items. A hierarchical block was built prior to the creation of the custom block to perform an equivalent function. The hierarchical block contained 34 library blocks. The "Separator (Simple)" block was subsequently built and replaced all 34 blocks originally used to perform the function. The custom block allowed for more output paths and was much more convenient to use. It was also about $1/25^{\text{th}}$ of the size (in number of bytes) as the original hierarchical block.

The "Activity, Delay (Waste)-testblk" in the *RYP Lib1* library is a custom block similar to the "Activity, Delay (Attributes)" block in the Extend Discrete Event library; this block delays items and interacts with attributes. The custom block can also generate a single waste stream from the process, which is represented by a separate output item carrying certain attributes. A hierarchical block performing an equivalent function was also built and tested against the custom block. A large model was constructed consisting of many of the hierarchical blocks being tested; few other blocks were used so that the contribution of the block in question would be apparent. The model with the hierarchical blocks was 3,086 kilobytes in size, required over 15 minutes to load, and took 52 minutes to run. With the custom blocks replacing the hierarchical blocks, the model file was only 32 kilobytes, required less than one minute to load, and executed in 1 minute 28 seconds. Using the custom block in place of the hierarchical block resulted in a model that was nearly 100 times smaller, loaded in less than $1/15^{\text{th}}$ of the time, and ran about 35 times faster.

In a large model where there are a variety of blocks, not all of which are custom, the speed and size improvements of customization will not be as dramatic as in the previous example. However, by using many different custom blocks, the combined small improvements achieved by each can collectively make a significant difference. For example, a model of plutonium operations focusing on weapon component fabrication was built and exercised. This model contained only standard Extend blocks and was over 2 megabytes in size. Many custom blocks from the *RYP Lib1* library were inserted in the model replacing several of the standard block structures. After the replacement process the model was one-half the original size and executed in less than one-half the time. Even in a large, complex model the collective contributions of individual custom blocks can be significant.

4.0 Conclusion

While the graphic nature of Extend makes it an easy tool to use, the real versatility of the software comes by way of customization through ModL scripting. The ability to create and modify Extend blocks allows a vast degree of extensibility and flexibility at a level not easily achievable with standard blocks. Customization can result in improved speed, size, and functionality of simulation models. While there are certain challenges to customizing Extend, the resulting capability gives the user a more powerful, efficient simulation tool.

References

Diamond, Bob, "Concepts of Modeling and Simulation," *PC AI*, September/October 1996, pp. 22-26.

ExtendTM: Performance Modeling for Decision Support, User's Manual for Extend, version 3 (1995).

Harrell, Charles and Tumay, Kerim, Simulation Made Easy: A Manager's Guide, Industrial Engineering and Management Press: Norcross, GA. (1995).

Appendix

The charts on the following page (Figure 1 and Figure 2) show the blocks from the *RYP Lib1* library (as of December 9, 1997). It also indicates the size that each block would be expected to contribute to a model created under the Windows operating environment (as taken from the "Get Info" window for each individual block - listed in number of bytes).

The five pages following the chart contain brief descriptions of each of the blocks shown. It is also noted whether the block can be used in Discrete Event (DE) models only, or whether they can also be used in Continuous (C) models as well. The descriptions assume familiarity with Extend.

Activity, Delay (Waste)-testblk	Activity, Multiple (Items/wk)	Add (5)	Animate Item (level)	Auditor
2586	1580	524	1130	1114
Batch (Demand)- (know count)	Batch (Sum)	Batch (Variable)- (know count)	Batch Attribute Combiner	Batching Tank (Attributes)
1216	1450	1234	762	6512
Batching Tank (Attributes)-size6	Count Blocks (class)	Distributor	Distributor, Multiple	Diverter, Random
5042	564	5246	4984	902
Diverter, Random (type)	Downtime (Calendar)-Con. Option	Exit Max AnimateH	File Input, Small (1)	File Input, Small (4)
1338	3420	1124	960	1714
Find Block (H)	Gate (multIV)	Generate Item/Clone	Get Batch Size- (know count)	Holding Tank (total)
29584	596	1262	804	652
Info Recorder	Info Recorder (Large)	Lookup Table (1)-(Attrib)	Lookup Table (4)-(Attrib)	Machine (D=R1+R2*X)
48756	48022	1294	2048	2006

Figure 1. Custom Blocks in the RYP Lib1 library

Machine (Type)	Machine [D=R1(t)+R2(t)*X]	Matching- (know count)	Pause (Item count)	Pause (time)
1336	2178	4472	584	558
Queue Gate (Attrib)	RealTimer (con)	Select DE Output (Attrib)	Select DE Output (Attrib) (5)	Separator (Simple)
1322	306	964	1044	3652
Set Attribute (1)	Set Attribute (short)	Spacer	Spring Switch	Subtract (5)
836	770	280	560	528
Sync. Gate	Sync. Gate (5)	Tag & Combine (5)	Throw (no table)	Unbatch (5)
1158	1346	988	824	1346
Unbatch (Variable)- (know count)				
1224

Figure 2. Custom Blocks in the RYP Lib1 library (continued)

Brief Descriptions of Blocks in the RYP Lib1 Library

Activity, Delay (Waste)-testblk: This block is an activity delay that interacts with attributes and can generate a single waste stream. [For DE models only]

Activity, Multiple (Items/wk): This activity block can work on many items at once. The one requiring the least time is output first. It has the added capability of calculating the delay time by entering the items per time unit or by reading the items per time unit from an attribute. [For DE models only]

Add (5): This block adds the values at the input connectors (up to 5 inputs). [For DE or C models]

Animate Item (Level): This block flashes when an item enters the block (like the normal Animate Item block does). It can also animate a level given a value between 0 and 1. [For DE models only]

Auditor: This block counts and tallies the number of items and specified attribute amounts that pass through a designated region of the model. [For DE models only]

Batch (Demand)-(know count): This block pulls in items until signaled, then releases them as one item. The batched output item is assigned information regarding how many items the batch contains. This info is stored in the item's itemArrayI[][2] slot (which is a user defined integer value). The Unbatch (variable)-(know count) block can use this information to determine how many items to break a batch into. [For DE models only]

Batch (Sum): This block allows items from several sources to be joined as a single item. The output item contains all the attributes from the joined input items (unless the *strip attributes* option is selected) with all like attribute values being summed. [For DE models only]

Batch (Variable)-(know count): This block pulls in a specified number of items then releases them as one item. The batched output item is assigned information regarding how many items the batch contains. This info is stored in the item's itemArrayI[][2] slot (which is a user defined integer value). The Unbatch (variable)-(know count) block can use this information to determine how many items to break a batch into. [For DE models only]

Batch Attribute Combiner: This block manipulates attributes and priorities on BATCHES by combining the attributes and priorities from the constituent items that make up the batch. [For DE models only]

Batching Tank (Attributes): This block accumulates specified attribute values from incoming items. It generates an item (batch), using only the specified attributes, when a certain batch size is accumulated. The attribute values assigned to the batch depend on the dialog options chosen. A batch can also be triggered by an outside connector signal. [For DE models only]

Batching Tank (Attributes)-size6: This block accumulates specified attribute values from incoming items. It generates an item (batch), using only the specified attributes, when a certain batch size is accumulated. The attribute values assigned to the batch depend on the dialog options chosen. A different batch size can be specified for each attribute listed. [For DE models only]

Count Blocks (class): This block counts the regular library blocks (by class) that are in a model. [For DE or C models]

Distributor: This block processes an item and distributes its attributes between the desired available outputs. It can interact with attributes prior to distribution. [For DE models only]

Distributor, Multiple: This block can work on many items at once. The block processes an item and distributes its attributes between the desired available outputs. The item requiring the least time is distributed and output first. [For DE models only]

Diverter, Random: This block diverts a certain fraction of the input items to the bottom output. The fraction represents the probability that an item will go out the bottom. The probability can be specified in the dialog or can be read from an attribute. [For DE models only]

Diverter, Random (type): This block diverts a certain fraction of the input items of a particular type to the bottom output. The fraction represents the probability that an item will go out the bottom. Probabilities for up to 10 different item types (which correspond to row numbers) can be entered in the dialog's data table. The item type is read from an attribute. Data can be entered manually into the table or read in from a text file. [For DE models only]

Downtime (Calendar)-Con. Option: This block schedules downtime based on dialog input. It has both item output and value output connectors (either of which can be used). [For DE models only]

Exit Max AnimateH: This block passes items out of the simulation and counts them. It allows the user to specify a maximum number of items that can exit through the block, at which point the simulation is stopped. It can also reflect its animation onto an enclosing hierarchical block. [For DE models only]

File Input, Small (1): This block provides data from a table to the model. The viewable table size is 10 rows by 1 column. Data can be entered manually into the table or read in from a text file. [For DE or C models]

File Input, Small (4): This block provides data from a table to the model. The viewable table size is 10 rows by 4 columns. Data can be entered manually into the table or read in from a text file. [For DE or C models]

Find Block (H): This block finds both normal and hierarchical type blocks by name. [For DE or C models]

Gate (multiV): This block restricts the number of items in a specific area of the model. The block properly accounts for multi-valued items. [For DE models only]

Generate Item/Clone: This block passes an incoming item through output "a" and generates either clones of the input item or blank items at outputs "b" and/or "c". Multiple quantities of items can be made. [For DE models only]

Get Batch Size-(know count): This block reads and outputs the batch size of passing batches (items) that were created using the Batch (Demand)-(know count) or Batch (Variable)-(know count) block. [For DE models only]

Holding Tank (total): This block accumulates input values, outputs an amount specified by the "want" connector, and displays the contents. It also displays the total contents the tank has seen over time. [For DE or C models]

Info Recorder: This block displays information about passing items and writes the data to a text file. There is a 350-item limit at which point the simulation stops. [For DE models only]

Info Recorder (Large): This block displays information about passing items and writes the data to a text file at the end of the simulation. It can handle very large numbers of items. [For DE models only]

Lookup Table (1)-(Attrib): This block looks up and outputs a particular row of data from the table (table size is 10 rows by 1 column). The row of data used corresponds to an attribute value on the input item. Data can be entered manually into the table or read in from a text file. [For DE models only]

Lookup Table (4)-(Attrib): This block looks up and outputs a particular row of data from the table (table size is 10 rows by 4 columns). The row of data used corresponds to an attribute value on the input item. Data can be entered manually into the table or read in from a text file. [For DE models only]

Machine ($D=R1+R2*X$): This block is a machine that operates on an item. The delay time used is calculated based on the parameters shown in the equation as set by the user in the block's dialog. Parameter values can be random or constant. [For DE models only]

Machine (Type): This block is a machine that operates on an item. An attribute identifying the item type number is read from the input item. The processing time used is the value listed in the table for the corresponding type number. [For DE models only]

Machine [$D=R1(t)+R2(t)*X$]: This block is a machine that operates on an item. The delay time used is calculated based on the parameters shown in the equation as defined by the user in the block's dialog. The R1 and R2 parameters are functions of the type (t) of item that enters. Different parameters can be used for different item types. [For DE models only]

Matching-(know count): This block matches items with equal attribute values and releases them as a batch when the conditions have been met. The batched output item is

assigned information regarding how many items the batch contains. This info is stored in the item's itemArrayI[][2] slot (which is a user defined integer value). The Unbatch (variable)-(know count) block can use this info to determine how many items to break a batch into. [For DE models only]

Pause (item count): This block pauses the simulation when the number of items to pass through the block reaches a certain number. [For DE models only]

Pause (time): This block pauses the simulation after a certain amount of simulation time. [For DE or C models]

Queue Gate (Attrib): This block is a queue with a gating feature that limits the amount of a specified attribute in a restricted section of the model. The block is a combination of a queue and a gate block. [For DE models only]

RealTimer (con): This block tracks the real-time duration of the simulation and outputs this time value at a connector. [For DE or C models]

Select DE Output (Attrib): This block selects one output from the two available. One of every N items goes to bottom connector unless the *select* connector is connected or unless the output is chosen based on the value of an attribute on the input item. [For DE models only]

Select DE Output (Attrib) (5): This block selects one output from the five available. It toggles outputs, unless the *select* connector is connected or unless the output is chosen based on the value of an attribute on the input item. [For DE models only]

Separator (Simple): This block separates an incoming item into multiple quantities of output items. The attributes on the incoming item can also be separated between the various output items. [For DE models only]

Set Attribute (1): This block assigns an attribute to passing items. (Only one attribute can be set at a time). [For DE models only]

Set Attribute (Short): This block assigns attributes to passing items. The block restricts the attribute name lengths to 15 characters. [For DE models only]

Spacer: This block passes items through. It is intended to be a "dummy" block that provides a space for a sensor (or other viewing type) connector when, otherwise, two viewing type connections would have to be made to the same connector (which is not allowed). [For DE models only]

Spring Switch: This block simulates a spring-loaded switch. It outputs a TRUE value for a specified duration when the Reset connector is activated. [For DE models only]

Subtract (5): This block subtracts the values at the connected inputs (up to 5 inputs). [For DE or C models]

Sync. Gate & Sync. Gate (5): These blocks only let the specified numbers of items proceed when the requirement at each input source is met. (The Sync. Gate has 3 inputs and 3 outputs. The Sync. Gate (5) has 5 inputs and 5 outputs.) [For DE models only]

Tag & Combine (5): This block combines items from various inputs into a single stream of items. It optionally tags items with an attribute identifying which input it came in on. It also allows the priority of the items passing through to be set to user specified values. [For DE models only]

Throw (no table): This block is essentially the same as the Throw block from the Discrete Event library, except that it has no table listing all the catch blocks in the model. [For DE models only]

Unbatch (5): This block separates an incoming item into multiple quantities of output items. The block has 5 output connectors. [For DE models only]

Unbatch (Variable)-(know count): This block clones an input item into multiple output items. The block has the option of reading the number of output items from the user defined itemArrayI[][2] slot on the input item. This can be used to unbatch items that were batched with any of the other “(know count)” blocks in the RYP Lib1 library. [For DE models only]

M98004310

Report Number (14) LA-UR--97-5032

Publ. Date (11) 199712

Sponsor Code (18) DOE/MA, XF

UC Category (19) UC-905, DOE/ER

DOE