Prototyping real time engineering systems using Hatley & Pirbhai's requirement model

Technical Report No 131

David A Fensome

April 1992

PROTOTYPING REAL TIME ENGINEERING SYSTEMS USING HATLEY & PIRBHAI'S REQUIREMENT MODEL

David A Fensome

School of Information Sciences, Hatfield Polytechnic,
Hatfield Herts AL10 9AB UK

Abstract A research programme is underway which is aimed at establishing ways of prototyping real time engineering systems using existing concurrent programming systems. The role of formal techniques in the development of prototypes, and its subsequent reuse in a well engineered product, is also an important theme. The first prototypes have used the Hatley & Pirbhai Transformation Schema, and have been built using Ada. A case study is described.
The research shows that a prototype based on Hatley & Pirbhai's requirement model would require a significant structural overhead, mainly because of the model 'execution' assumptions. Also Ada is an inappropriate language to use, mainly because the tasking overhead is too great, but also because the communications model differs from Hatley & Pirbhai's model. Some future research direction are indicated.

Keywords Control system design, Computer simulation, Modelling, Prototyping, Real time computer systems, Software development, Software specification, Transformation Schema.

INTRODUCTION

A research programme is currently underway which is aimed at establishing ways of prototyping real time engineering systems. In particular

- To evaluate the suitability of existing concurrent programming

systems

- To establish the role of formal techniques in the development of prototypes
- To identify ways that this formality could subsequently be reused in the development of a well engineered software product.
- To provide an environment to improve the prototyping process.

Prototyping in the context of this research means the examination of areas of uncertainty in the functional requirements by execution of a software model. Other aspects of prototyping such as target system performance and user interface considerations are regarded as separate problems, and are not considered. Real time engineering systems for this purpose covers that class of systems that are event driven, are (software) process oriented, and concerned with the control and monitoring of equipment in its widest interpretation (sometimes called 'reactive' systems). See (Harel, 1992) for a general discussion on modelling and prototyping of reactive systems.

The initial work is to build prototypes from requirements specifications expressed in natural language and diagrams. The first prototypes have used the Hatley & Pirbhai Transformation Schema (Hatley, 1988), since this is based on well understood and widely used data flow techniques from Structured Systems Analysis (Yourdon, 1989), and have been implemented in Ada. The rationale for this strategy lies in the intuitive notion that mapping from a data flow model to a concurrent procedural language (Ada in this case) should be easy. Earlier work using Transformation Schema as a basis for prototypes can be found in (Coomber, 1990) where Smalltalk is used as a process definition language, and a Petri net execution approach has been used.

THE CROOK PROOF VENDING MACHINE CASE STUDY

The first prototype was built using the Hatley & Pirbhai requirement model of the 'Crook Proof Vending Machine' (CPVM), and an Ada animation. The

requirements statement and top level modelling is described in Chapter 28 of (Hatley, 1988), and briefly, is an attempt to build a vending machine that does not allow customers to fool the machine into giving product when it should not. The machine also has facilities for storing information on product quantity and price.

As the model was built assumptions were made about the mechanisms the system had to interface into, since these were not available in (Hatley, 1988), and were important in deriving the model. For example it was assumed each input coin fell into a coin hopper where physical measurements were made on the coin (producing the_object: PHYSICAL in Fig 1), and the coin was then routed to the appropriate coin register, or returned to the user as a 'slug'.

Extracting the data models using the requirement dictionary given in (Hatley, 1988) was hard, mainly because emphasis is given to flow (and data store) 'types' eg group/primitive, internal/external, control/data, discrete/continuous, whereas the 'attributes' name /units /range /resolution /rate contain most of the information on data types. In fact both objects and types appear as entries in the requirement dictionary, whereas separate entries in a types and a data dictionary proved much more helpful in building the prototype.

USING THE REQUIREMENT MODEL AS A PROTOTYPE MODEL

Intuitively it would seem that the two models should be the same and the animation could be coded directly from the requirement model. However there are some problems with this.

Model Execution

The requirement model is built with the intention of 'mental execution', with only fine grain execution (almost single step mode), and a lot of assumptions about the execution environment. Firstly, as a consequence of the infinitely fast execution of the processes in the requirement model, there is no interleaving of events and each event is processed atomically. The prototype model would have to guarantee

this by serial feedback on completion of event processing, or by some overall parallel monitoring of completion of data flow. Also the process model is only feed forward, and any accidental feedback via a complex process network will not produce deadlock or instability (or the mental execution environment will 'sort it out'). Finally the activation/deactivation of sets of processes by the associated control model must be completed after the atomic processing of an event, and this therefore implies a separate control phase in the prototype model.

Data Flow

The requirement model data types require access mechanisms in the prototype model eg a flow of 'read_price' (see Fig 2) requires to be changed to a new type 'price X product' for the prototype model. Also continuous data flow requires special attention in the prototype model (see below), and data stores in general require mutual exclusion mechanisms for readers and writers.

The requirements dictionary with standard BNF like data dictionary notation was found to be satisfactory for a high level view of the system. However refinement of types and objects using the Z type system (Potter, 1991), and separate dictionaries for types and objects, allowed easy translation to Ada.

Finally data flow arrows entering a requirement model process do not necessarily correspond to Ada task entries. Sometimes an Ada task will be in the 'server' role (with entries), and sometimes active calling other tasks. A better graphical model based on Buhr's notation (Buhr, 1984) was found to be much more convenient for the prototype.

USING ADA AS AN ANIMATION TOOL

The requirement model uses hierarchical process decomposition which is cumbersome to emulate with Ada's tasking model. This is because each level in the requirement model requires an Ada task with a 'select' and multiple 'accepts' just to pass on data to lower levels in the hierarchy.

```
package body context_data is

        task body top_level is
        begin
                loop
                        select
                                accept the_object (.....) do
                                  level1_data.validate_coins.the_object ()
                                end the_object;
                        or
                                accept etc
```

Fig. 3. An Ada implementation of a top level requirement process

Also the possibility of deadlock with this structure is very high because of the additional rendezvous. This could be avoided by 'flattening' the process network, but this then means that large requirement models become potentially unmanageable. Also the positive advantage of automatic consistency checking between levels by the compiler (the 'balancing' part of the requirement model build), would be lost.

This structural overhead, and associated deadlock risk, could be significantly reduced using OCCAM with named channels as parameters to lower level procedures (representing parallel processes), as shown in Fig. 4 below.

```
PROC level0 (CHAN OF INT coin,
             CHAN OF REAL32 price.....

PAR
             level1.1 (coin, ....)
             level1.2 (price, ....)

PROC level1.1 (CHAN OF INT coin, .....)
```

```
PAR
        level1.1.1 (coin,
        level1.1.2 (....
```

Fig. 4. Skeleton OCCAM process structure using channels

However the data typing facilities offered by OCCAM are significantly poorer than those of Ada.

Another problem using Ada is that the Ada task state is not the same as a requirement model process state. In the requirement model a process is activated and deactivated by the underlying control model (a finite state machine), which effectively removes any communication with a deactivated process. In Ada a task runs forever, or until terminated in some way, and then completes. This means that entries have to be provided to activate and deactivate an Ada task, and in addition, to be able to check the state of a task using a complex select/loop structure for each requirement model process. A possible structure for each requirement model process is shown below.

```
accept start
--first start
end
loop
        select
                accept entry1    --data flow 'port'
                        -- process it
                end
        or
                accept entry2
        or
                accept stop
                end stop
                loop
                        select
                                accept start
                                        exit loop
                                end
                        or
                                accept is_go (..)
                                end
                        end select
                end loop
        or
                accept is_go (..)
                end
        end select
end loop
```

Fig. 4. Possible process state emulation with Ada.

The Ada communication model is also different to the requirement model in several ways. Firstly communication is not via a named channel, but rather by a named port (the entry), giving the cumbersome Ada task structure mentioned above. Also Ada tasks must check whether a consumer task is active before communication is started, otherwise deadlock will occur. In addition it is desirable to include a single buffer in an Ada emulation, at least to decouple producer and consumer, but also to provide a simulation of a continuous data flow (ie to allow random re-reads of a data flow by a consumer process).

Finally there is no direct equivalent of the split or merge data flow operations that occur in the requirement model. These would have to be provided by yet another Ada task.

These communications differences indicate that a generic package is required for the Ada emulation, containing one task for each data flow, and generic parameters defining the data flow type and the entry name of the consuming task. This was built for unidirectional data flow, as shown in Fig. 5, but the Ada language would not allow a generic split or merge.

```
generic
        type MESSAGE is private;
        --data flow type
        with procedure fire_entry (the_mess : in
                    MESSAGE);
        --consumer entry name
        with function is_go return BOOLEAN;

        --process  state check

package uni_df is
        procedure put_df (fwd_mess : in MESSAGE);
        function get_df return MESSAGE;
end uni_df;
```

```
package body uni_df is

        buffer : MESSAGE;
        function get_df return MESSAGE is
        begin
                return buffer;
        end get_df;

        task buff_server  is
                entry put ( input : MESSAGE);
        end buff_server;

        task body buff_server  is
                begin
                loop
                        accept put ( input : MESSAGE) do
                                buffer := input;
                        end put;
                        if is_go then fire_entry (buffer);
                        end if;
                end loop;
        end buff_server;

        procedure put_df (fwd_mess : in MESSAGE) is
        begin
                buff_server.put  (fwd_mess)
        end put_df;
end uni_df;
```

Fig. 5. Ada Uni-directional data flow emulation

CONCLUSIONS

Using Hatley and Pirbhai's requirement model as a basis for a prototype model is not quite so attractive as intuition might indicate, mainly because of the underlying 'execution' assumptions described above. In fact a prototype based on this model would require a significant structural overhead, almost like a conventional discrete event simulation package.

Using Ada to animate the prototype has shown that Ada is not an appropriate language, mainly because the tasking overhead is too great, and there is ample opportunity for deadlock. The tasking overhead occurs when implementing the requirement model process hierarchy, data flow and process state. One reason for this overhead is that the Ada communication model is many to one 'ported', whilst named channels are used in the requirement model. To some extent animations based on programming languages using channels, such as OCCAM, would be better, as long as the data modelling and generic facilities are reasonably good.

For these reasons it is not felt to be productive to pursue using Hatley and Pirbhai's requirement model as a prototype model, or its Ada animation.

Then next stage of the research will focus on a more appropriate (perhaps more abstract) requirement model without all the execution assumptions implicit in Hatley and Pirbhai's requirement model. It is felt that implementing prototypes from such models using conventional programming languages will be easier. Candidate requirement models are a Z model (Potter, 91) and a UNITY model (Chandy, 88), both based on mathematical models at a relatively high level of abstraction. However the issue of non determinism in these specifications, and how this is dealt with in the prototype, will have to carefully considered. The UNITY model has some advantage here, since the model is based on nondeterministic selection of statements executed in parallel and constrained by a fairness rule.

REFERENCES

Buhr, R. J. A. (1984) *System Design with Ada* Prentice Hall

Chandy and Misra (1988) *Parallel Program Design* Addison Wesley

Coomber, C. J. and Childs, R.E. (1990) *A Graphical Tool for the prototyping of Real Time Systems* ACM Sigsoft Vol15 No 2 Apr 1990

Harel, D. (1992) *Biting the Silver Bullet* IEEE Computer Jan 1992

Hatley, D.J. and Pirbhai, I. A. (1988) *Strategies for Real-Time System Specification* Wiley

Potter, B, Sinclair, J and Till, D (1991) *An Introduction to Formal Specification and Z* Prentice Hall

Yourdon, E. (1989). *Modern Structured Analysis* Prentice Hall

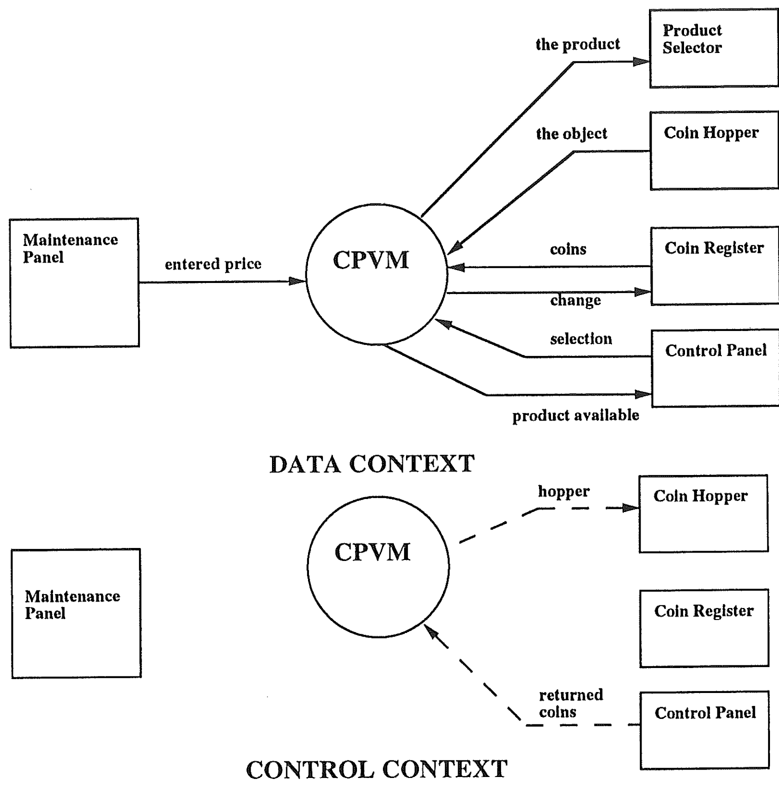


Fig. 1 Crook Proof Vending Machine

(after Hatley & Pirbhai)

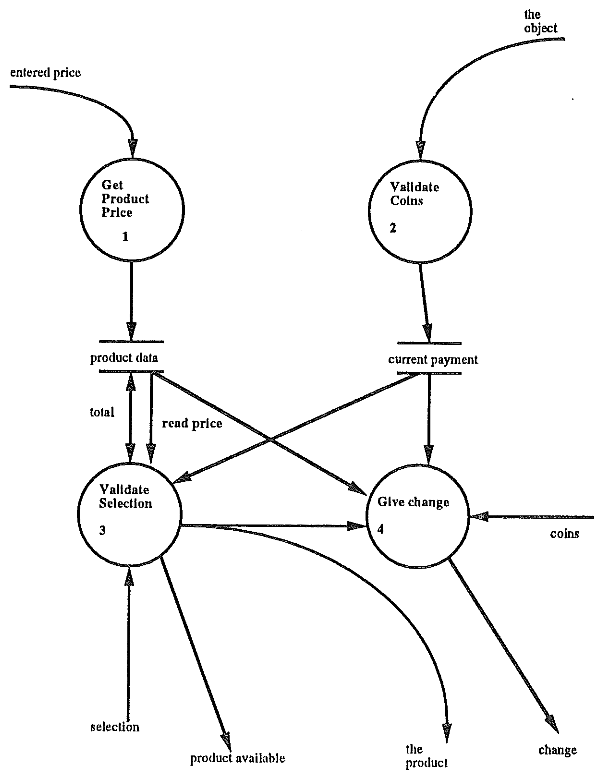
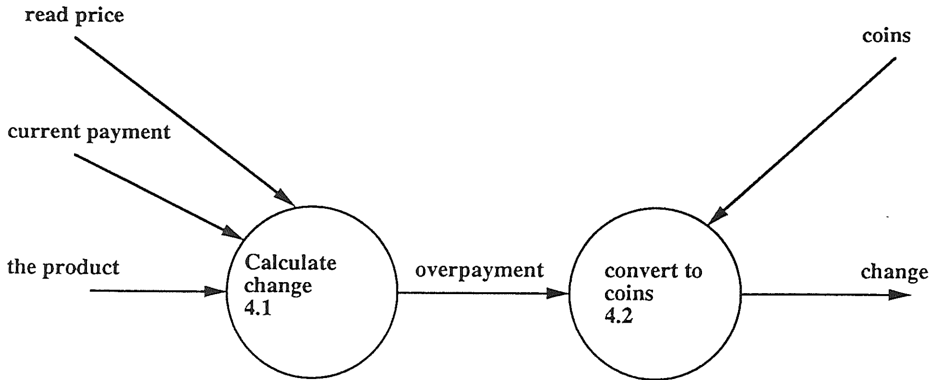
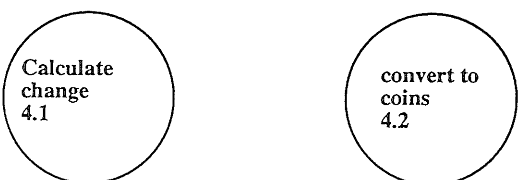
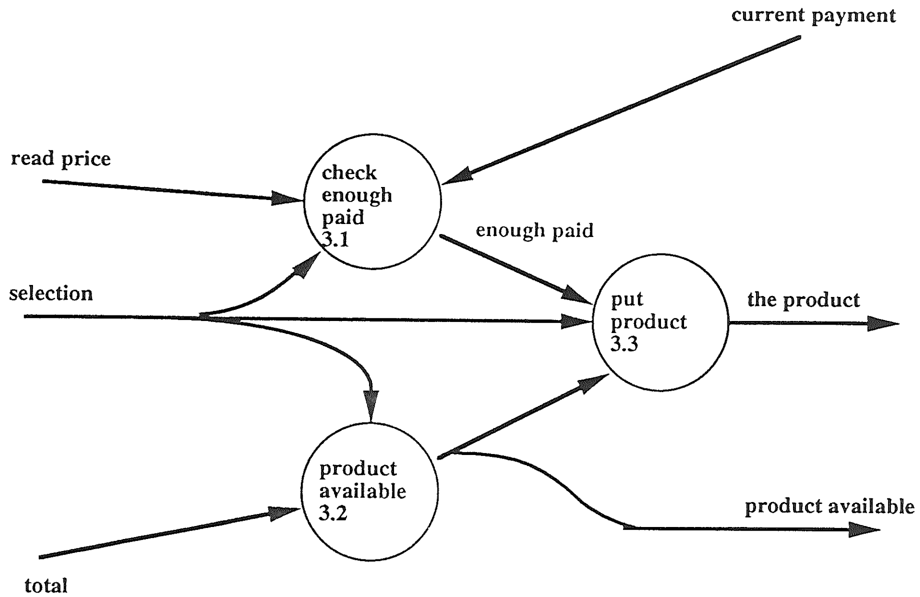


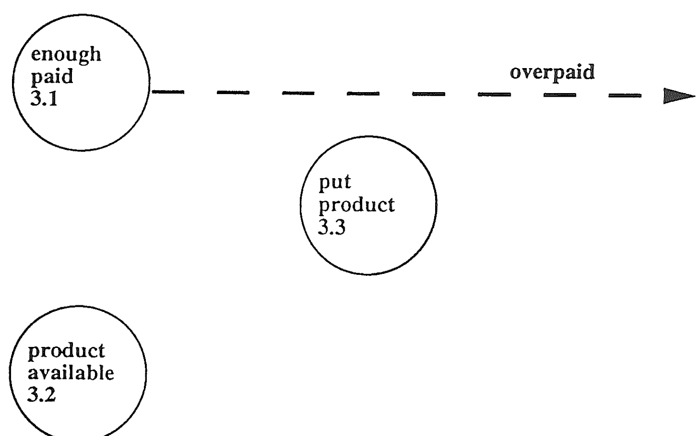
Fig. 2 CPVM DFD 0 Level 1



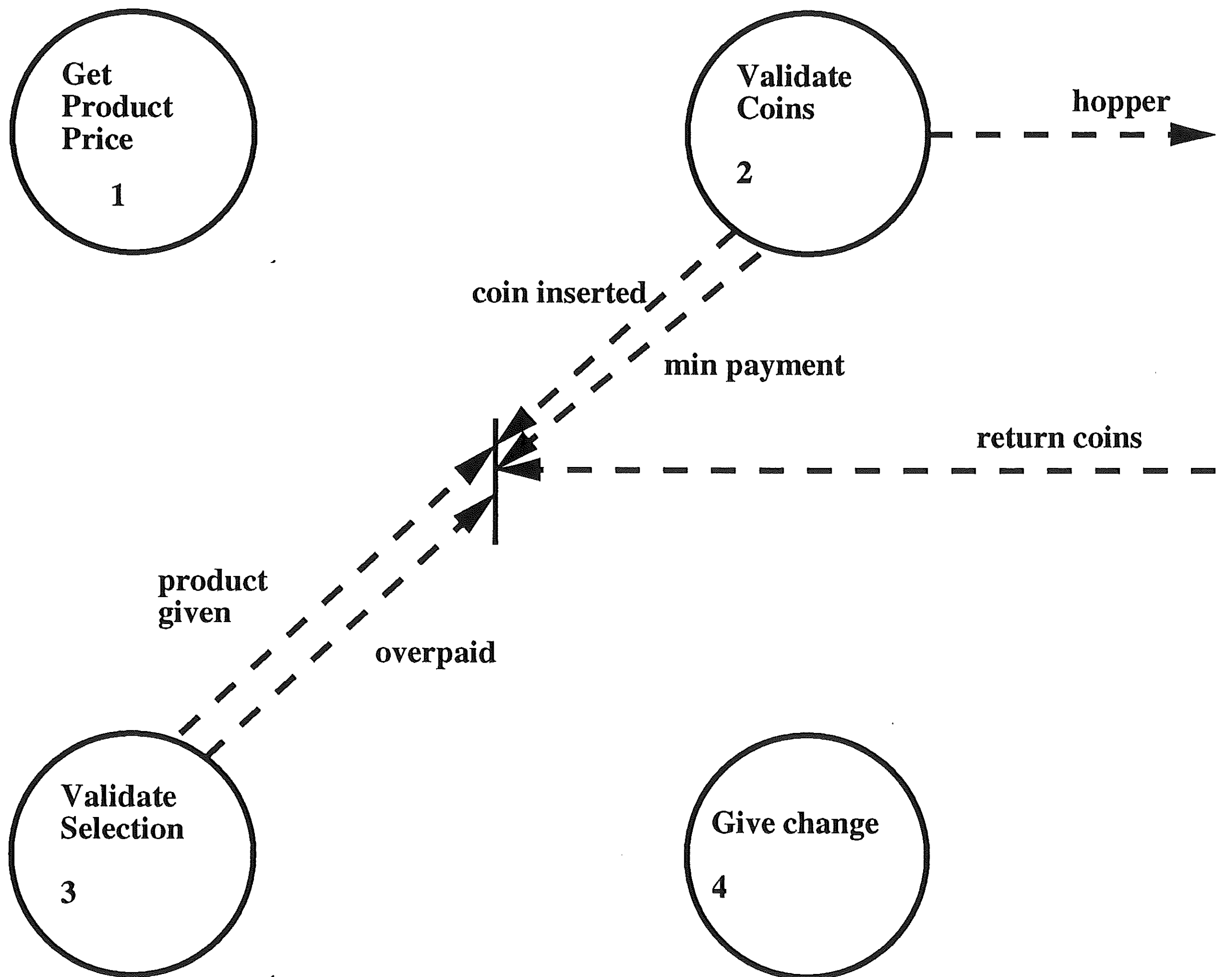


dfd/cfd 4 level 2





DFD/CFD 3 level 2



CPVM CFD 0 Level 1

Flow/variable	Type	Reference	comments
change	COINHOPPER	context	assumes mechanism can give n coins of each type
coin_inserted	EVENT	cfd0	not a 'slug' input
coins	COINHOPPER	context	from the mechanical coin register
current_payment	MONEY	cfd0	total customer legal coin payment
enough_paid	BOOLEAN	dfd3	current_payment >= cost
entered_price	CATALOGUE	context	maintenance input data
hopper	OBJECT	context	controls input coin hopper
min_payment	EVENT	cfd0	customer can now select goods
overpaid	EVENT	cfd3	customer has paid too much money
overpayment	MONEY	dfd4	requires change to this amount
product_data	P.CATALOGUE	dfd0	product information store
product available	BOOLEAN	context	light on the control panel
read_price	COST	dfd0	product price read from product data
return_coins	EVENT	context	customer wants all coins back
selection	PRODUCT	context	data from the control panel
the_object	PHYSICAL	context	data represents object inserted
the_product	PRODUCT	context	data to the product selector
total	GOODSHOPPER	dfd0	total number of product items left

Crook Proof Vending Machine - Data dictionary

Type	definition	comments
CATALOGUE	PRODUCT X COST X GOODSHOPPER	
COINCOUNT	0..Maxcount	
COINHOPPER	COINCOUNT X COINCOUNT X COINCOUNT	50ps X 20ps X 10ps
COST	Mincost..Maxcost	
GOODSHOPPER	0 .. Maxhopper	
MONEY	Minmoney..Maxmoney	
OBJECT	10p|20p|50p|slug	
PRODUCT	Beer | Cola	assume just 2 products
PHYSICAL	DIAMETER X WEIGHT	
DIAMETER	Mindia .. Maxdia	
WEIGHT	Minweight .. Maxweight	
EVENT	null	no values

Crook Proof Vending Machine -Types Dictionary