Formalization and enforcement of requirements to modular discrete-event simulation runtime

E. V. Chemeritskiy, K. O. Savenkov

This paper presents an extendable architecture for a discrete-event simulation runtime (DESR). The architecture is based on a set of logic blocks. Each block encapsulates a part of the DESR functionality and provides an interface to that functionality for other blocks. Differences in requirements that are imposed by different simulation problems are encapsulated in distinct logic blocks. Interface of each block is formally specified and there is a possibility of its automated check. Instances of the logic blocks are combined to get DESR for a particular simulation problem. Therefore, there is no need either in performance trade-offs or in a custom development of the DESR.

General Terms: **Discrete-Event Simulation, Simulation Runtime, Reuse, Modular Design**

I. Introduction

THIS paper is devoted to the development of the architecture for a DESR consisting of a set of logical blocks. The particular set of blocks, their interfaces and functionality give ability to build the runtime taking into account the requirements of a custom simulation task.

The Computer Systems Laboratory of CMC MSU conducts multiple diverse research projects related to the simulation of distributed systems. The terms of such problems include the modeling of on-board systems (aviation, naval, automotive) computer networks and instruction set of processors. All these problems are focused on modeling the functionality of a computer system (data processing) and its performance as well as used apparatus – discrete-event simulation.

In CSL, such problems are solved since 1982 and with increasing number of projects and directions it was decided to create a unified DESR [1]. It is based on a single approach to the computer systems simulation and uses a specialized language to describe simulation models.

However it became clear that this approach is not entirely suitable for the development of high-tech research projects. New challenges bring with them the need for simulation in a variety of detail levels. New requirements for scalability, performance and response time appear. General purpose solution is a compromise between expressive power and efficiency. Another problem is that it requires a tremendous effort to upgrade and maintain it in the grease condition in the future development.

As a result incompatible changed have been made in the unified DESR for each major project and currently there are several well-used variants of the same product.

This paper proposes the other way - to develop the architecture of the DESR as a collection of logical blocks. Each of these blocks encapsulates the functionality of the DESR and provides an interface to other units to use this functionality. A set of blocks has been designed so that the differences in the requirements for the DESR made by different tasks were encapsulated in separate blocks, retaining the overall structure of the DESR. The option to compose a DESR from needed copies of the different blocks gives ability to create a DESR configured to solve a custom simulation task with no need to compromise in terms of system performance and without wasting the developers' effort to create and support a new DESR.

This paper is based on a comparative analysis of different editions of the runtime DYANA used in several projects: 1) functional simulation of on-board marine systems [3], 2) hardware-in-the-loop simulation of on-board aircraft systems [4], 3) simulation of performance of neuroprocessor instruction set [5] and 4) on-board automotive information system simulation. Several DESR of well-known simulation systems have been also reviewed: AutoMod, SLX, Extend, SIMAN V, ProModel, GPSS/H.

The research results into a set of blocks encapsulating the differences of the examined DESR. On the basis of the proposed blocks a mathematical model of the DESR has been constructed. The paper describes in detail the functionality of each of the proposed block and the formal specifications of the interfaces of these blocks. Some mechanisms for its automated check have been proposed according to the analysis of designed specifications.

II. Components of the runtime

A generalized DESR scheme is proposed on the basis of a comparative analysis of different variants of DYANA runtime [2]-[5] and several well-known simulation systems: AutoMod, SLX, Extend, SIMAN V, ProModel, GPSS/H [6], [7].

The terms of this paper are borrowed from [8] with some

Manuscript received March 22, 2010.

E. V. Chemeritskiy is with the Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow, (e-mail: tyz@lvk.cs.msu.su).

K. O. Savenkov is with the Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow ,(Phone: +7(495)939-46-71; fax: +7(495)939-25-96; e-mail: savenkov@cs.msu.su).

generalization. The basic concepts of generalized DESR are *event* (a signal notifies DESR on the changing of model state), *logical object* (LO) (entity that is able to schedule events) and *resource* (provide LO with some services). Resources are presented by model time, the cells of the memory (variables), clipboard information, semaphores, and so on. LO may delay event arrival until some condition depending on a state of the model resources set is satisfied. This condition is called a *delay condition*. As a result of the event arrival DESR can produce a number of actions called the *event handling*.

All event transactions take place in the *handling blocks*. Each block is a container for events generated by LO. Any handling block can be provided with an individual scheduler to properly rank the elements of the block. Runtime environment may contain several types of handling blocks:

1. *Current event block* (CEB) contains ready to be handled events.
2. *Future event block* (FEB) stores events with a delay condition depending only on the model time.
3. *Delayed event block* (DEB) contains events with complex delay condition depending on set of resources. There are two types of DEB: *related event block* (REB) and *polled event block* (PEB).

The work of DESR blocks is coordinated by the *dispatcher* throughout the model execution. The result message sequence is written to output *trace*.

More information about logical blocks composing the runtime environment and their functionality within the DESR is available in [9].

III. Static Semantics

A. Static structure of the runtime

Runtime environment w connects a set of resources $\mathbb{R}$ and logical objects $\mathbb{L}$ of the model with the dispatcher d.

$$\mathrm{w} =< R, L, d > \tag{1}$$

It is important to note that the resources and the logical objects are binded to the only dispatcher throughout the model execution.

B. Resource

The set of all resources is denoted by $\mathbb{R}$. For each resource $r \in \mathbb{R}$ is attached a set of variables (memory cells) $V \subseteq \mathbb{V}$ and event container E. The variables from the set $\mathbb{V}$ can take values from the set $\mathbb{D}$.

There are two types of resources: direct access resources $\mathbb{R}_{direct}$ and indirect access resources $\mathbb{R}_{indirect}$. Direct access resource keeps track of the values of related variables using the map $\mathcal{E}: \mathbb{R}_{direct} \to \mathbb{D}$. In this case the resource stores only a "local copy" of the original model data. Over a set of indirect access resources a map $\mathcal{D}_R: \mathbb{R}_{indirect} \to \{d, \emptyset\}$ to the set of dispatchers is defined. This mapping allows one to distinguish between attached and free model resources.

$$r =< V, E > \tag{2}$$

$$\mathcal{E}: \mathbb{R}_{direct} \to \mathbb{D} \tag{3}$$

$$\mathcal{D}_{\mathrm{R}}: \mathbb{R}_{indirect} \to \{d, \emptyset\} \tag{4}$$

C. Message and trace

The output trace t is presented by the message sequence. The message alphabet is denoted by $\mathbb{M}$.

$$t = \{\mathrm{m}_k\}_{k=1}^{\infty} \tag{5}$$

D. Event

Suppose there exists a set of all possible events $\mathbb{E}$. Each event has a trace message m and the type of event p. There are several event types in accordance with type of handling blocks intended to store this event $p \in \{CEB, FEB, PEB, REB\}$.

There are a number of maps defined over event set $\mathbb{E}$. The logical object arisen the particular event could be found by the mapping $\mathcal{L}: \mathbb{E} \to \mathbb{L}$. Suppose that there is a set of event attributes of all kinds $\mathbb{A}$. Then the mapping $\mathcal{A}: \mathbb{E} \to 2^{\mathbb{A}}$ defines an attribute set for each particular event.

Suppose there is a set of predicate symbols $Pred$ defined over a set of resources $\mathbb{R}$ AND depending on the values D of attached variables V. So the delay condition of event $e \in E$ could be presented as a formula of propositional logic (quantifier-free first-order logic) $\mathcal{C} \in Cond$ over the set of predicates $Pred$.

The map changing the states set of resources $\Re$ and the states set of logical objects $\mathfrak{L}$ associated with the event $e \in \mathbb{E}$ is referenced as the modification $\mathcal{M}: \Re \times \mathfrak{L} \to \Re \times \mathfrak{L}$. There are only several ways to change model state:

1. To attach new logical object $l \in \mathbb{L}$ to the dispatcher d,
2. To attach new resource $r \in \mathbb{R}$ to the dispatcher d,
3. To change resource variables value $r.V$,
4. To change logical object activity limit value $\bar{a}$.

$$e =< m, p, \mathcal{C}, \mathcal{M}, \mathcal{A} > \tag{6}$$

$$\mathcal{L}: \mathbb{E} \to \mathbb{L} \tag{7}$$

Event type imposes restrictions on the delay condition. The delay condition of events of type "CEB" is always true. Events of type "FEB" essentially depends only on the model time and types "PEB" and "REB" by contrast are independent of model time.

The notation $Var(\phi)$ denotes a set of significant variables of $\phi \in Cond$ and T denotes a resource containing the model time. Then, the following expressions are correct [1]:

$$\forall e\ (e.p = CEB \to Var(e.\mathcal{C}) = \emptyset) \tag{8}$$

$$\forall e\ (e.p = FEB \to Var(e.\mathcal{C}) = T) \tag{9}$$

$$\forall e\ ((e.p = PEB \vee e.p = REB) \to T \notin Var(e.\mathcal{C})) \tag{10}$$

The delay condition of each "FEB" event is true at a single

[1] Here and henceforth the operator "." will be used to denote the tuple element.

point on the axis of time.

$$\forall e\ \exists!\ D \in \mathbb{D}\ (e.p = FEB \rightarrow e.\mathcal{C}(D) = true) \quad (11)$$

Introduce a special operator $\mathcal{T}: \mathbb{E} \rightarrow \mathbb{D}$ to determine its value.

$$\mathcal{T}(e) = D \in \mathbb{D}: \mathcal{C}(D) = true \quad (12)$$

There is also a dependency between the type of event and its attribute set. The events of one type have the same attribute set. Attribute set of the event with a type different from "CEB" includes all of its attributes.

$$\forall e_1 \forall e_2 \left(e_1.p = e_2.p \rightarrow E\left(\mathcal{A}(e_1)\right) = E\left(\mathcal{A}(e_2)\right)\right) \quad (13)$$

$$\forall e_1 \forall e_2 \left(e_1.p = CEB \rightarrow E\left(\mathcal{A}(e_1)\right) \subseteq E\left(\mathcal{A}(e_2)\right)\right) \quad (14)$$

E. Logical object

The logical object $l \in \mathbb{L}$ uses the event generator $g \in \mathbb{G}$to schedule events. For each generator $g \in \mathbb{G}$ its current state $s \in \mathbb{S}$ and the step function $\mathcal{N}: \mathbb{S} \rightarrow \mathbb{S} \times \mathbb{E}$ are defined.

Newly scheduled events are stored in the local event queue $E = \{e_i \in \mathbb{E}\}$ [2] with size limited by the capacity $c \in \{\mathbb{N}, 0\}$. In addition to capacity the behavior of the logical object is controlled by the activity level $a \in \{\mathbb{N}, 0\}$ and the activity limit $\overline{a} \in \{\mathbb{N}, 0\}$.

Over the set of logical objects the mapping $\mathcal{D}_L: \mathbb{L} \rightarrow \{d, \emptyset\}$ to the dispatcher set is defined. This mapping allows one to distinguish between attached and free logical objects.

$$g =< s, \mathcal{N} > \quad (15)$$

$$l =< g, E, a, \overline{a}, c > \quad (16)$$

$$\mathcal{D}_l: \mathbb{L} \rightarrow \{d, \emptyset\} \quad (17)$$

Generator $g \in \mathbb{G}$ constructs an event sequence $\{e_n\}_1^\infty$ using the recurrence relation $s_{n+1} \times e_{n+1} = g.\mathcal{N}(s_n)$. Generator is scheduling events in order of nondecreasing model time.

$$\forall n, k \in \mathbb{N}\ (n > k \rightarrow \mathcal{T}(e_n) \geq \mathcal{T}(e_k)) \quad (18)$$

The generator has ability to synchronize its own time (presented explicitly or implicitly, through the scheduled events) with the model time. In this case the step function returns an empty symbol as the event $s_{n+1} \times \theta_e = g.\mathcal{N}(s_n)$. If the generator has planned all the events then step function returns an empty symbol as the state $\theta_s \times e_{n+1} = g.\mathcal{N}(s)$.

F. Handling block

Handling block consists of one or several event containers and a scheduler $\mathcal{S}: 2^{\mathbb{E}} \rightarrow \{e_i\},\ \mathcal{S}(\mathbb{E}) \subseteq \mathbb{E}$. The scheduler ranks elements of attached containers and choose a certain event range.

[2] We assume that the event queue E has several predefined operations:
1. $|E|$ – returns number of events in the queue,
2. $pushBack(E,e)$ – adds event to the back of queue,
$popFront(E)$ – takes event from the head of queue.

There are several types of handling blocks b: “CEB", "FEB", "PEB" and "REB”. Each block type has its own distinctive features.

Blocks typed as "CEB", "FEB" and "REB" have a more complex structure than a block of type "PEB". Two containers are attached to these blocks. Block "FEB" also contains a model clock (defining the event horizon) and the simulation threshold l. "REB" block has a resource container R_{REB} intended to store resources changed on the current iteration of event handle loop.

$$b_{\text{CEB}} =< E_{CEB}, \hat{E}_{CEB}, \delta_{CEB} > \quad (19)$$

$$b_{\text{FEB}} =< E_{\text{FEB}}, \hat{E}_{\text{FEB}}, \delta_{\text{FEB}}, l > \quad (20)$$

$$b_{\text{PEB}} =< E_{\text{PEB}}, \delta_{\text{PEB}} > \quad (21)$$

$$b_{\text{CEB}} =< E_{\text{REB}}, \hat{E}_{\text{REB}}, \delta_{\text{REB}}, R_{\text{REB}} > \quad (22)$$

The event scheduler gives as the result an ordered set of events with the delay condition met. The results of the scheduler of "CEB" and "FEB" blocks includes all such events whereas the schedulers of the remaining block types allowed not giving all such events.

$$\forall X \in \{CEB, FEB, PEB, REB\}\ \forall d \in \mathbb{D}$$
$$\forall e \in \delta_X(E)\ e.\mathcal{C}(d) = true \quad (23)$$

$$\forall X \in \{CEB, FEB\}\ \forall d \in \mathbb{D}\ \delta_X(E) = \{e: e.\mathcal{C}(d) = true\} \quad (24)$$

$$\forall X \in \{PEB, REB\}\ \forall d \in \mathbb{D}\ \delta_X(E) \subseteq \{e: e.\mathcal{C}(d) = true\} \quad (25)$$

The delay condition of event moved to "FEB" block essentially depends on the model time, and is satisfied at only point on the time axis. Scheduler of this block gives a set of events with a minimum time:

$$\forall e \in \delta_{FEB}(\mathbb{E})\ (\mathcal{T}(e) = \min_{e \in \mathbb{E}} \mathcal{T}(e)) \quad (26)$$

In addition time of each scheduled event does not exceed the simulation threshold l:

$$\forall e \in \delta(\mathbb{E})\ \mathcal{T}(e) \leq l \quad (27)$$

G. Dispatcher

Output trace t, handling blocks set B and direct access resource container R_{direct} are attached to the dispatcher d.

$$d =< t, B, R_{direct} > \quad (28)$$

Handling block set includes a single block of “FEB”, and can also include no more than one block of every other type. Thus the assertion

$$\{b_{FEB}\} \subseteq B \subseteq \{b_{CEB}, b_{FEB}, b_{PEB}, b_{REB}\} \quad (29)$$

IV. Operational semantics

In this chapter the concept of runtime environment and rules of its changing are introduced. Algorithms for the initialization of the runtime and model running are described.

A. Resource state

For each resource $r =< V, E >$ the state $\mathfrak{r}$ is determined as a tuple consisting of variable value $r.V$ [3] and a set of events from attached container $.E$ [4].

$$\mathfrak{r} =< eval(V), eval(E) > \tag{30}$$

B. Logical object state

The state $\mathfrak{l}$ of the logical object $l =< g, E, a, \bar{a}, c >$ is defined as a pair of corresponding event generator $l.g$ and the event set of events contained in the attached to the logical object container $l.E$.

$$\mathfrak{l} =< s, eval(E) > \tag{31}$$

C. Handling block state

State $\mathfrak{B}$ of handling block set (29) is defined as a collection of the contents of containers attached to them.

$$\mathfrak{b}_{\text{CEB}} =< eval(E_{\text{CEB}}), eval(\hat{E}_{\text{CEB}}) > \tag{32}$$

$$\mathfrak{b}_{\text{FEB}} =< eval(E_{\text{FEB}}), eval(\hat{E}_{\text{FEB}}) > \tag{33}$$

$$\mathfrak{b}_{\text{PEB}} =< eval(E_{\text{PEB}}) > \tag{34}$$

$$\mathfrak{b}_{\text{REB}} =< eval(E_{\text{REB}}), eval(\hat{E}_{\text{REB}}), eval(R_{\text{REB}}) > \tag{35}$$

D. Dispatcher state

Dispatcher state $\mathfrak{d}$ is defined as a collection of the contents of trace t, attached handling blocks state $\mathfrak{B}$ and the content of attached resource container.

$$\mathfrak{d} =< eval(t), \mathfrak{B}, eval(R_{direct}) > \tag{36}$$

E. Runtime environment state

Runtime environment condition ϖ is characterized by the state of resources $w.\mathbb{R}$ and logical object $w.\mathbb{L}$ of the model and the state of the dispatcher $w.d$.

$$\varpi =< \mathfrak{R}, \mathfrak{L}, \mathfrak{d} > \tag{37}$$

F. The rules of the runtime state changing

Rules (38)-(40) are intended to add a direct or indirect access resource or logical object to the dispatcher.

Rule (41) describes the changes in the state of the runtime after the value of resource variables changed.

Rule (42) defines the run of the model.

$$\frac{r \in \mathbb{R}_{direct} \,\&\, r \notin d.R_{direct}}{d.R_{direct} = r \cup d.R_{direct}} \tag{38}$$

$$\frac{r \in \mathbb{R}_{indirect} \,\&\, r.d \neq d}{r.d = d} \tag{39}$$

$$\frac{l \in \mathbb{L} \,\&\, l.d \neq d}{l.d = d, lookForEvents(l)} \tag{40}$$

$$\frac{r \in \mathbb{R}_{indirect} \,\&\, set(r, D)}{r.D = D, R_{REB} = r \cup R_{REB}} \tag{41}$$

$$\frac{E_{\text{FEB}} \neq \emptyset}{handleCycle(), advanceTime()} \tag{42}$$

G. Searching events to transit into the handling blocks $lookForEvents(l \in \mathbb{L})$;

The algorithm checks the readiness of the logical object to schedule events and moves events created by them into the handling blocks attached to the dispatcher. If the number of such events in the handling blocks has reached the limit, some of them are buffered into the local queue.

#Used while event handling – decrement activity level
$a = a - 1$;
IF ($a = 0$)
  #There are no more events produced by l
  IF ($|E| = 0$)
    #Local event queue is empty
    WHILE ($s \neq \theta_s$ & $(a < \bar{a} \vee |E| < c)$) DO
      #Event generator can schedule events and local queue is not full
      #Invoke event generator
      $s \times e = \mathcal{G}(s)$;
      IF ($e = \theta_e$)
        #Generator cannot schedule events yet
        BREAK;
      FI
      IF ($a < \bar{a}$)
        #Activity level is less than activity limit
        #Add event to handling block
        $addEvent(e)$;
        #Increment activity level
        $a = a + 1$;
      ELSE
        #Handling block contain a limit event number
        #Add event to local event queue
        $E.pushBack(e)$;
      FI
    OD
  ELSE
    #Local event queue is not empty
    #Add a number of events less or equal to activity limit
    $limit = min(|E|, \bar{a})$;
    WHILE ($a < limit$) DO
      #Transmit event from local queue to handling blocks
      $addEvent(E.popFront())$;
      $a = a + 1$;
    OD
  FI
FI

[3] To indicate the values of the object X the operator $eval(X)$ will be used.
[4] Under the value of the container the set of elements contained therein is infered.

H. Addition of event to the handling blocks

$addEvent(e \in \mathbb{E})$;

The event is placed in a handling block in accordance with its type.

#Load an event to handling block of the same type
IF ($p = CEB$)
  $\hat{E}_{CEB} = \hat{E}_{CEB} \cup e$;
ELSEIF ($p = PEB$)
  $E_{PEB} = E_{PEB} \cup e$;
ELSEIF ($p = REB$)
  $\hat{E}_{REB} = \hat{E}_{REB} \cup e$;
#Event type is "FEB" – other choices are sort out
ELSEIF ($t = c$)
  #Event time is equal to current model time
  $\hat{E}_{FEB} = \hat{E}_{FEB} \cup e$;
ELSEIF ($t > l$)
  #Event time is greater than simulation threshold
  $l.s = \theta_s$;
ELSE
  $E_{FEB} = E_{FEB} \cup e$;
FI

I. Event handle cycle

$handleCycle()$;

Algorithm iteratively retrieves ready events from handling blocks, sorts them and calls for event handling. Model time does not change during this process.

WHILE (TRUE) DO
  #Search for direct access resources with changed variable value
  $\forall r \in R_{direct}$
    IF ($eval(r.V) \neq \mathcal{E}(r)$)
      #Add resource to container of changed resource
      $R_{REB} = r \cup R_{REB}$;
      #Refresh value
      $r.V = r.\mathcal{G}$;
    FI
  #Compose the contents of REB major event container
  #Add events with satisfied delay conditions
  $E_{REB} = \{e : e \in \hat{E}_{\mathrm{REB}} \ \& \ e.\mathcal{C} = true\}$
  #Add the rest to resource containers
  $\forall e \in \hat{E}_{\mathrm{REB}} \setminus E_{REB}$
    $\forall r \in Var(e.\mathcal{C})$
      $r.E = e \cup r.E$;
  #Leave in auxiliary REB container only events with satisfied delay condition
  $\hat{E}_{\mathrm{REB}} = E_{REB}$;
  #Add events depending on changed resources into the REB major event container
  $\forall r \in R_{\mathrm{REB}}$
    $E_{\mathrm{REB}} = r.E \cup E_{\mathrm{REB}}$;
  #Add events independent on resource state into CEB auxiliary container
  $\hat{E}_{\mathrm{CEB}} = \hat{E}_{\mathrm{CEB}} + \hat{E}_{\mathrm{FEB}}$;
  #Compose the contents of CEB major event container
  $E_{\mathrm{CEB}} = \hat{E}_{\mathrm{CEB}} + \delta_{\mathrm{PEB}}(E_{\mathrm{PEB}}) + \delta_{\mathrm{REB}}(E_{\mathrm{REB}})$;
  $E = \delta_{\mathrm{CEB}}(E_{\mathrm{CEB}})$;
  IF ($|E| = 0$)
    #No event was chosen
    BREAK:
  FI
  #Handle chosen events
  $\forall e \in E$
    $handleEvent(e)$;
OD
$R_{REB} = \emptyset$;

J. Event handling

$handleEvent(e)$;

Handled event is excluded from the handling block. Then the model state changed in appropriate to this event way and the information recorded in the trace. The dispatcher also searches for the events created by the same logical object to transfer into attached handling blocks.

IF ($p = CEB \vee p = FEB$)
  #Event does not depend on resource state
  $\hat{E}_{CEB} = \hat{E}_{CEB} \setminus e$;
ELSEIF ($p = PEB$)
  $E_{PEB} = E_{PEB} \setminus e$;
ELSEIF ($e \in \hat{E}_{REB}$)
  #Event delay condition was always satisfied
  $\hat{E}_{REB} = \hat{E}_{REB} \setminus e$;
ELSE
  #Event was added into resource container
  $\forall r \in Var(e.\mathcal{C})$
    $r.E = r.E \setminus e$;
FI
#Change model state according to event
$\Re \times \mathfrak{L} = e.\mathcal{M}(\Re \times \mathfrak{L})$;
Add message to the output trace $e.m$;
$getEvents(L(e))$;

K. Model time advancing

$advanceTime()$;

Selected by the scheduler of "FEB" block, events are transferred from the major container to the additional one. During this model time is changed to the arrival time of any of the selected events (each of these events has the same arrival time).

#Move event scheduled to current model time to auxiliary container FEB
$\hat{E}_{FEB} = \delta_{FEB}(E_{FEB})$;
$E_{FEB} = E_{FEB} \setminus \hat{E}_{FEB}$;
#Set a new model time
$c = e.t, e \in \hat{E}_{FEB}$;

V. Requirements to the runtime blocks

Mathematical model allows identifying interfaces of blocks that make up the runtime. Thus a sufficient condition for the possibility of substituting a new runtime is the compliance of its interfaces with the requirements.

There are several classes of requirements:

1. PRED – specifies pre-condition,
2. POST – specifies post-condition,
3. RET – determines the method return value,
4. EQ – specifies equivalence to the described algorithm.

Requirements for the interfaces of the blocks depend on its type, and the configuration of the runtime as a whole. All specifications are listed in table 1.

VI. Observations on the implementation

Configurable runtime environment is developed as a compile time library written in C++. The flexibility of the runtime components is achieved through the use of template classes. Thus each of these blocks is represented as a single class. The new runtime environment with the necessary properties can be created on the basis of the class layout.

The signature of class interfaces are variable and depend on the configuration of the runtime. Nevertheless, they can be checked at compile time. As appropriate tool the library Boost Concept Check Library (BCCL) can be used [10]. This library allows formal describing of the requirements for abstract data types (concepts) used in templates and verify their compliance with these requirements.

The most difficult requirements to verify are the ones to interface of the handling blocks. Depending on the configuration of the runtime their functionality can have significant differences. But the number of fundamentally different configurations of the handling block is low. Thus partially specifying a template of dispatcher class and using BCCL can impose restrictions on the interfaces of the handling blocks for any possible configuration.

Semantic requirements for class interfaces can be checked with unit testing. Tests for the blocks can be incorporated directly into the library being developed so that using the predefined flag tests new plug-in logical block [11]. Then successfully tested blocks can be incorporated into the library itself.

VII. Conclusion

Implementation of the developing library results into ability to quickly build high-performance runtime environment with the necessary properties. This only requires new instances of some blocks. Thus a sufficient condition for the correctness of the constructed DESR is the interface compliance to formulated specifications which can be verified automated.

Tested blocks can in turn be included into the library. With the increasing number of block instances the share of reusable code will increase whereas the cost of developing new runtime will be reduced to the layout of the ready-made blocks.

References

[1] V. G. Moloney, R. L. Smelyansky, "An integrated approach to modeling distributed computing systems", Programming N.1, 1988 pp. 57-67 (In Russian)

[2] A. Bakhmurov, A. Kapitonova, R. Smeliansky, "DYANA: An Environment for Embedded System Design and Analysis", 5-th International Conference TACAS'99, Amsterdam, The Netherlands, March 22-28, 1999. Springer (LNCS Vol.1579), pp.390-404

[3] V. V. Balashov, A. G. Bahmurov, D. Yu. Volkanov, R. L. Smelyanskiy, M. V. Chistolinov, N. V. Yushchenko, G. T. Mamontov, P. Yuhta "Experience of the program DYANA implementation for simulation and integration of on-board computing systems", Abstracts of reports XXVI conference in memory of an outstanding designer gyroscopic devices N. N. Ostryakov - St. Petersburg: Central Research Institute Elektropribor, 2008. pp. 60-61 (In Russian).

[4] V. V. Balashov, A. G. Bakhmurov, M. V. Chistolinov, R. L. Smeliansky, D. Yu. Volkanov, N. V. Youshchenko, "A Hardware-in-the-Loop Simulation Environment for Real-Time Systems Development and Architecture Evaluation", In Proc. of the Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX 2008, Szklarska Poreba, Poland, June 26-28 2008.

[5] A. G. Bahmurov, E. G. Egisapetov, O. V. Novikov, V. V. Prus, K. O. Savenkov, R. L. Smelyansky, "Tool support for software development process for a special processor-based CPU L1879VM1 ", Methods and means of processing information. Proceedings of the Second All-Russian Scientific Conference. - M.: Publishing Department, Faculty of Computational Mathematics and Cybernetics. Moscow State University, 2005, pp.450-456

[6] T. J. Schriber, D. T. Brunner, "Inside discrete-event simulation software: how it works and why it matters", Proceedings of the Winter Simulation Conference, 2005, pp. 11 pp.+

[7] T. J. Schriber, D. T. Brunner, "Inside discrete-event simulation software: how it works and why it matters", Proceedings of the Winter Simulation Conference, 1996, pp. 11 pp.+

[8] P.J. Sanchez, "Fundamentals of simulation modeling", Proceedings of the Winter Simulation Conference, 2007, 9-12 Dec. 2007 Page(s):54 – 62

[9] K. O. Savenkov, E. V. Chemeritskiy, "Discrete-event simulation runtime: from genericity to extendability and reuse", Simulation-2010, submitted for publication.

[10] Boost library [Online]. Available: http://www.boost.org.

[11] A. H. Bagge, V. David, M. Haveraaen. "The axioms strike back: testing with concepts and axioms in C++", In Proceedings of the Eighth international Conference on Generative Programming and Component Engineering (Denver, Colorado, USA, October 04 - 05, 2009).

Table 1. Requirements to components of the runtime.

Block	Method	Block type	Configuration	Requirement	
				Type	Specification
Resource	$get(void):D \in \mathbb{D}$			RET	$eval(r.V)$
	$check(void):\{true, false\}$	r_{direct}		RET	$\mathcal{E}(r) \neq eval(r.V)$
	$set(D \in \mathbb{D}):void$	$r_{indirect}$		POST	$r.V = D$
			Related waiting	PRED	$r.bind$ has been invoked.
				POST	$d.changed$ has been invoked.
	$bind(d):void$			POST	$r.d = d$
	$addEvent(e \in \mathbb{E}):void$			POST	$r.E = e \cup r.E$
	$deleteEvent(e \in \mathbb{E}):void$			POST	$r.E = r.E \setminus e$
	$getEventsRange(void):< first, last >$			RET	$< r.E.first, r.E.last >$
Trace	$addEvent(m \in \mathbb{M}, t \in \mathbb{D}):void$			POST	The message has been added to the trace.
Event	$getType(void):\{CEB, FEB, PEB, REB\}$			RET	$e.p$
	$getMessage(void):\mathbb{M}$			RET	$e.m$
	$getLogicalObject(void):\mathbb{L}$			RET	$\mathcal{L}(e)$
	$change(void):void$			POST	$\Re \times \mathfrak{L} = e.\mathcal{M}(\Re \times \mathfrak{L})$
	$getTime(void):\mathbb{D}$	e_{FEB}		RET	$\mathcal{T}(e)$
	$condition(void):\{true, false\}$	e_{PEB} e_{REB}		RET	$e.\mathcal{C}$
	$addToResources(void):void$		Related waiting	POST	$\forall r \in Var(e.\mathcal{C})\ r.E = e \cup r.E$
	$removeFromResources(void):void$			POST	$\forall r \in Var(e.\mathcal{C})\ r.E = r.E \setminus e$
Logical object	$bind(d):void$			POST	$l.d = d$
	$lookForEvents(void):void$			EQ	To algorithm $lookForEvents(l)$
	$setCapacity(c \in \{\mathbb{N}, 0\}):void$		Event buffer	POST	$l.c = c$
	$setLimit(\bar{a} \in \{\mathbb{N}, 0\}):void$			POST	$l.\bar{a} = \bar{a}$
Handling block	$addEvent(e \in \mathbb{E}):void$			POST	$b = e \cup b$
	$removeEvent(e \in \mathbb{E}):void$			POST	$b = b \setminus e$
	$getEventRange(void):< first, last >$			RET	$eval(b.\mathcal{S}(\{e: e \in b\}))$.
	$addIndependentEvents(< first, last >):void$	b_{CEB}	$b_{CEB} \in d.B$	POST	$b_{CEB} = \{< first, last >\} \cup b$
	$addEvents(< first, last >):void$	b_{CEB}	$b_{CEB} \in d.B$	POST	$b_{CEB} = \{< first, last >\} \cup b$
		b_{FEB}	$b_{CEB} \notin d.B$	POST	$b_{FEB} = \{< first, last >\} \cup b$
	$timeAdvance(void):void$	b_{FEB}		POST	$b_{FEB}.c = \min_{e \in b_{FEB}} \mathcal{T}(e)$
	$addResource(r \in \mathbb{R}):void$	b_{REB}	$b_{REB} \in d.B$	POST	$b_{REB} = r \cup b_{REB}$
		b_{FEB}	$b_{REB} \notin d.B$ Related waiting	POST	$b_{FEB} = r \cup b_{FEB}$
	$reset(void):void$	b_{PEB}		POST	Resource container has been reset.
		b_{FEB}		POST	Ready to handle events have been reset.
		b_{CEB}		POST	Unhandled events have been reset.
Dispatcher	$addResourceDirect(r \in \mathbb{R}_{direct}):void$			EQ	To rule (1)
	$addResourceIndirect(r \in \mathbb{R}_{indirect}):void$		Related waiting	EQ	To rule (2)
	$resourceChanged(r \in \mathbb{R}):void$		$b_{REB} \in d.B$	POST	$b_{FEB}.addResource(r)$ has been invoked.
			$b_{REB} \notin d.B$ Related waiting	POST	$b_{FEB}.addResource(r)$ has been invoked.
	$addLogicalObject(l \in \mathbb{L}):void$			EQ	To rule (3)
	$addEvent(e \in \mathbb{E}):void$			EQ	To algorithm $addEvent()$
	$handleEvent(e \in \mathbb{E}):void$			EQ	To algorithm $handleEvent()$
	$handleCycle(void):void$			EQ	To algorithm $handleCycle()$
	$run(void):void$			EQ	To rule (5)