A GUI Approach to Programming of TMO Frames and Design of Real-Time Distributed Computing Software

K. H. (Kane) Kim and Seok-Joong Kang

Dream Lab, University of California
Irvine, CA 92697, USA
{khkim, seokjook} @uci.edu,   http://dream.eng.uci.edu/

Abstract

An advanced high-level approach for programming of real-time distributed computing applications, the TMO (*Time-triggered Message-triggered Object*) programming and specification scheme, has been enabled without creating any new language or compiler. Instead, a middleware system named the *TMO Support Middleware* (TMOSM) and an API that wraps around the execution support services of TMOSM have been established. An approach enabling further reduction of the labor in TMO programming is to let the programmer use a GUI to build structural frames of application TMO networks. The supporting tool called the *Visual Studio for TMO* (ViSTMO) consists of a GUI part for interactive design of TMO-network structures and a part for automated generation of C++ TMO code-frameworks. The TMO scheme contains mechanisms enabling efficient design of autonomy-rich structures of application systems and ViSTMO provides GUIs for programming the use of those mechanisms among others. The recent expansion and refinement of the functionality of and the implementation techniques used in ViSTMO are discussed in this paper.

Keywords: TMO, real time, time-triggered, computing, autonomy, GUI, automated generation, code-frameworks, ViSTMO.

1. Introduction

Programming of real-time (RT) distributed computing application software is nowadays a steadily growing branch of software engineering [Kim97, OMG01, OMG02, Sch00, Sel00]. Such programming has often been an order-of-magnitude harder than the long-practiced programming of non-real-time programs involving no network communications. To rectify this situation, new-generation methods and tools need to be established in sound forms enabling abstract styles of programming without depriving from the programmer essential abilities for controlling timely interactions and data flow among constituent program parts.

The *Time-triggered Message-triggered Object* (TMO) programming scheme is among the advanced high-level approaches to RT distributed programming [Kim97, Kim00, Kim02]. The support tools for the TMO

scheme can be based on well-established object-oriented (OO) programming languages such as C++, C#, and Java and on ubiquitous commercial RT OS kernels including derivatives of Linux or even on the Microsoft Windows family of OS kernels. TMO is a natural, syntactically small, and semantically powerful extension of the conventional object model(s) and typical OO programmers can adopt it with relatively small efforts. The TMO scheme enables structuring of every conceivable distributed computing application in the form of a TMO network. Also, the TMO scheme contains mechanisms enabling efficient design of autonomy-rich structures of application systems [Kim95, Kim00].

Moreover, TMO structuring supports various phases of system engineering, from abstract designs to detailed implementations. In addation, TMO is an effective mechanism not only for variable-degree abstraction of distributed RT systems under design but also for variable-accuracy RT simulation of the application environments [Kim97].

We have been enabling TMO programming without creating any new language or compiler. Instead, a middleware system named the *TMO Support Middleware* (TMOSM) and an API that wraps around the execution support services of TMOSM have been provided [Kim99, Kim00]. This API was named the *TMO Support Library* (TMOSL). Also, during the past two years we have been working on a GUI tool for interactive design of TMO-network structures and automated generation of code-frameworks to simplify the jobs of TMO programmers. The potential benefits of using GUIs in constructing large-scale programs were indicated by numerous researchers in the past [Mol01, Nie00, Ros02, Sof].

The GUI tool that is evolving in the authors' lab has been named the *Visual Studio for TMO* (ViSTMO). ViSTMO is a graphic-design-oriented tool that helps TMO designers and programmers to build application TMO networks efficiently. ViSTMO provides GUIs for programming autonomy-rich structures of application systems among others. The first prototype that demonstrates the feasibility and potential benefits of ViSTMO in a convincing manner has been implemented. Our experiences with ViSTMO have revealed that this GUI-based code-framework generation approach can lead to significant improvement of efficiency in designing and programming of application TMO networks. The field of high-level RT distributed programming is a young one, let

1

alone the field of GUI-based RT distributed programming. Therefore, we expect that the functionality of ViSTMO will continue to evolve for some years to come.

In this paper, the three most recent expansions of the functionality of ViSTMO are discussed. They are:
1) generation of code-frameworks linking a client to a server TMO when the user draws a line from the icon representing the client to icon representing the server;
2) packaging of all the code-frameworks of the TMOs to be hosted on a node into a single loadable package when the user graphically assigns a set of TMOs to a node, i.e., draws lines linking the icon representing a node to all the icons representing the TMOs to be hosted on that node; and
3) support for design of and generation of code-frameworks for logical multicast channels which can be used for loose coupling among various TMOs.

These enable considerable savings of the efforts of TMO network programmers. The techniques for implementing the parts of the code-frame generator related to these new features are also discussed.

This paper is organized as follows. A brief overview of the basic features of the TMO programming scheme is first given in Section 2. The features enabling efficient design of autonomy-rich structures is reviewed in Section 3. Then, the recent expansion and refinement of functionality of and the implementation techniques used in ViSTMO are discussed in Section 4. Section 5 concludes the paper.

2. Basic features of the TMO programming and specification scheme

The TMO scheme was established in early 1990's with the skeleton of a concrete syntactic structure and execution semantics to support economical reliable design and implementation of RT distributed computing systems. It has been enhanced in several steps since then.

TMO is a natural, syntactically simple, and semantically powerful extension of the conventional object(s) [Kim97, Kim00]. As depicted in Figure 1, the basic TMO structure consists of four parts:

ODS-sec = object-data-store section: list of *object-data-store segments* (ODSS's); Each ODSS is a group of data members and is a unit that can be locked for exclusive use by one method execution at a time as well as for shared use by multiple concurrent method executions which perform read-only operations on the data members contained.

EAC-sec = *environment access-capability* section: list of *gate* objects providing efficient call-paths to
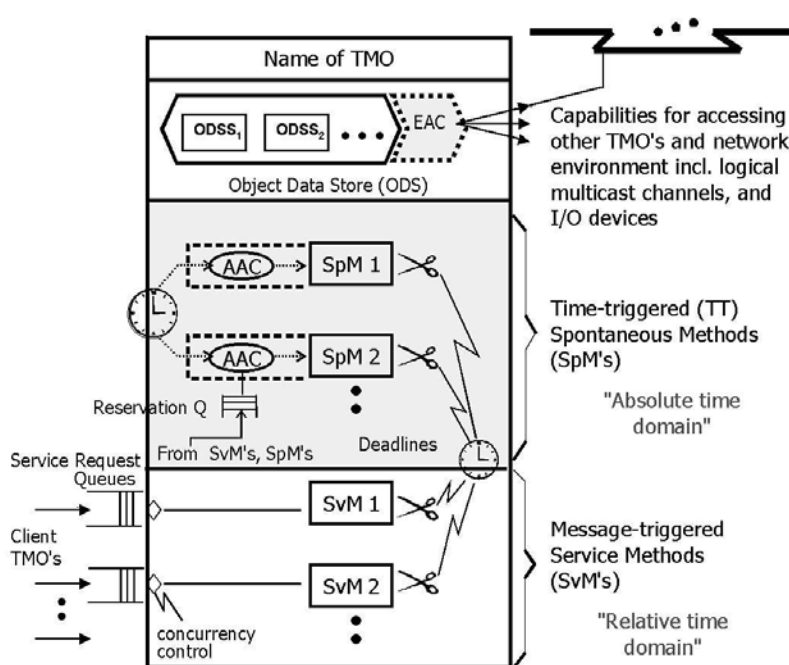

Figure 1. The Basic Structure of TMO (Adapted from [Kim97])

remote object methods, logical communication channels, and I/O device interfaces;

SpM-sec = *spontaneous-method* section: list of *spontaneous methods*;

SvM-sec = *service-method* section.

Major features are summarized below.

(a) Distributed computing component: The TMO is a distributed computing component and thus TMOs distributed over multiple nodes may interact via *remote method calls*. To maximize the concurrency in execution of client methods in one node and server methods in the same node or different nodes, client methods are allowed to make *non-blocking* types of service requests to server methods. Moreover, the designer of client methods may impose deadlines for result returns.

(b) Clear separation between two types of methods: The TMO may contain two types of methods, *timetriggered (TT-) methods* (also called the *spontaneous methods* or *SpMs*), which are clearly separated from the conventional *service methods* (*SvMs*). The SpM executions are triggered upon reaching of the RT clock at specific values determined at the design time whereas the SvM executions are triggered by calls from clients which are transmitted by the execution engine in the form of service request messages. Moreover, actions to be taken at real times which can be determined at the design time, can appear only in SpMs.

(c) Basic concurrency constraint (BCC): This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, *activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions*

2

www.manar

are not in place. An SvM is allowed to execute only when an execution time-window big enough for the SvM that does not overlap with the execution time-window of any SpM that accesses the same ODSSs to be accessed by the SvM, opens up. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(d) Guaranteed completion time of the server (i.e., an SvM of a server TMO) and the *result return deadline* imposed by the client.

Triggering times for SpMs must be fully specified as constants during the design time. Those RT constants appear in the first clause of an SpM specification called the *autonomous activation condition* (AAC) section. An example of an AAC is "for t = from 10am to 10:50am every 30min start-during (t, t+5min) finish-by t+10min" which has the same effect as {"start-during (10am, 10:05am) finish-by 10:10am", "start-during (10:30am, 10:35am) finish-by 10:40am"}.

3. Structuring of highly autonomous real-time distributed computing systems as TMO networks

The autonomy of various subsystems in complex RT distributed computing systems is a highly desired attribute. Highly autonomous subsystems enable concurrent and largely independent testing/verification and maintenance of the subsystems. A challenge in designing distributed computing systems is then to maximize the degree of autonomy of the subsystems while achieving the objective of efficient cooperative computing.

The TMO scheme contains several mechanisms enabling efficient design of highly decentralized application systems.

3.1 Service timing autonomy and temporal firewall

TMO methods have guaranteed completion times (GCTs) as their attributes. Additionally, SpMs have declarations of start time-windows. These make the worst-case timings of services provided by TMOs to be highly predictable.

SpMs are highly autonomous in the sense that their executions start in the absence of any client's requests and are not interfered by SvM executions under the BCC. Moreover, one can structure a TMO in which SvMs play the roles of "receptionists" which receive service requests and then convert them into work orders to be put in a queue in the ODS which is examined regularly by a "master" SpM. Essentially, the master SpM does the substantive computing work requested by the clients. In such a TMO, the master SpM has some freedom in sorting the queue of work orders and choosing the start time of handling each order. Such a TMO possesses a certain degree of *service timing autonomy* [Kim95].

The TMO scheme is aimed for enabling design-time guaranteeing of an *end-to-end delay bound*, i.e., a bound on the time interval from the instant at which a significant input event (e.g., a significant sensor value or message from the application environment) occurs to the instant at which a corresponding output action (e.g., an actuator command or a database update) does. GCTs of SvMs are known to and used by the designers of client TMO methods in deriving the GCTs of the client methods. Therefore, the designer of an application TMO network can visualize how a worst-case end-to-end delay will be composed of the GCTs of various contributing TMO methods.

The features mentioned above enable highly concurrent design and implementation of multiple TMOs and relatively easy composition of integrated TMO networks.

3.2 Relocation autonomy and Real-time Multicast and Memory-replication Channel (RMMC)

TMOs possess *relocation autonomy*, i.e., autonomy in choosing their locations. A TMO execution engine consists of a group of networked computing node platforms (hardware nodes plus OS kernels) and instantiations of the TMO Support Middleware (TMOSM) running on the node platforms. The location of each TMO is a global knowledge within the TMO execution engine. TMOs and SvMs are referenced by their location-independent global names. If a TMO is relocated, other TMOs which depend on the services of the relocated TMO are not impacted except for possible changes in the response times of the relocated TMO.

Another mechanism in the TMO scheme provides further options for exploiting autonomy-rich structures. In addition to the interaction mode based on remote method invocations, TMOs can use another interaction mode in which messages can be exchanged over logical message channels of which access gates are explicitly specified as data members of involved objects. The advanced type of such channel facility adopted in the TMO scheme is called the *Real-time Multicast and Memory-replication Channel* (RMMC) [Kim00], of which an earlier version was called the HU data field channel [Kim95] and was in turn an extension of the data filed channel [Mor82, Mor86].

For example, access gates for two RMMCs (RMMC1 and RMMC2) can be declared as data members of each of the three remotely cooperating RT objects (TMO1, TMO2, and TMO3) during the design time. Once TMO1 sends a message over RMMC1, the message will be delivered to the buffer allocated inside the execution engine for each of the three RT objects. Later during their execution, certain methods in TMO2 and TMO3 can pick up those messages by sending the requests through their RMMC1 gates to their execution engines. In many applications, this interaction mode leads to better efficiency than the interaction mode based on remote

3

method invocations.

An RMMC can be implemented over point-to-point networks as well as over broadcast-enabled bus networks. Many RMMCs can be multiplexed on a given physical communication network.

In case an RMMC is dedicated to the carrying of requests for a particular type of services and relevant results being returned, the client TMOs which are connected to the RMMC and are users of the services, do not even need to know the names of the server TMOs or their SvMs. A client TMO can just load a request with appropriate parameters onto the RMMC and later pick up the results from the RMMC. Therefore, there is further autonomy on the part of the server TMOs in such an application TMO network.

The RMMC scheme supports not only conventional event messages but also state messages based on distributed replicated memory semantics [Kop97]. A state message carries information to be stored in a fixed memory location in each receiver corresponding to the ID of the state message.

A state message's ID represents a group of replicated memory units, each capable of holding the information carried in the state message and belonging to a different receiver. A state message producer timestamps the message at message-production time. Each receiver reads the content of its state message memory through a relevant gate at a convenient time. This means that the producer may update the contents of the state message memory units at a higher frequency than the frequency at which a certain receiver reads the content of its state message memory. A state message is thus typically used to share the periodically observed state information about a dynamic state-varying item, like a car's position. The state message mechanism facilitates loose coupling between senders and receivers.

4. ViSTMO : Architecture and implementation techniques
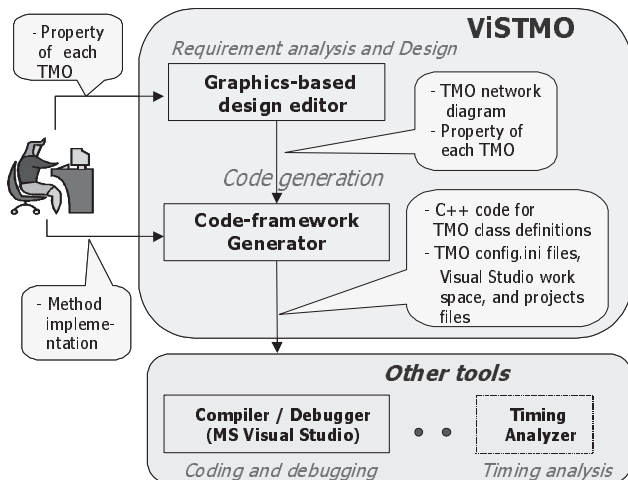

Figure 2. Major components and functionality of ViSTMO

ViSTMO provides the *Graphics-based Design Editor* that supports interactive design of application TMO networks. It also has the *Code-framework Generator* that generates code-frameworks based on the information provided by a programmer using the Graphics-based Design Editor. Figure 2 shows major components and functionality of ViSTMO.

4.1. Graphics-based Design Editor

ViSTMO provides user-friendly GUIs for application TMO network programmers. Primarily by filling the empty fields in dialog boxes, programmers can produce designs of application TMO networks that are complete except for function bodies of object methods. For example, a programmer can add/revise/delete information about various parts of a TMO using the dialog box depicted in Figure 3. Such information represents the properties of the TMO. The screen in Figure 3 shows that the TMO component, Radar, which is to be realized by instantiating the automatically named class, CRadar, contains among other things an SpM, SpM0_CRadar. If the user selects SpM0_CRadar and pushes the button "Revise" in Figure 3, then a window that allows the user to specify some detailed properties of SpM0_CRadar opens up.

ViSTMO can also check certain design errors or design inconsistencies and report them to the programmer. The important advantages of using the Graphics-based Design Editor of ViSTMO are as follows.

(1) Minimize the amount of data which a programmer should input: A programmer does not need to create codes for the default constructors of ODSS classes and TMO classes. Once the programmer inputs the essential parameters of ODSS and TMO classes, the default constructors for them can be generated by ViSTMO. In the case of building an SpM, the essential parameters that the user needs to input are: SpM name, the selections (via mouse) of the ODSSs to be used and the access modes (Read only / Read and Write / No access) of the SpM for the selected ODSSs, timing specification (i.e., AAC), the selections (via mouse) of the remote SvM call capabilities to be used and the types of service calls to be made, and the selection (via mouse) of the multicast channels (i.e., RMMCs) to be used. Then ViSTMO will generate C++ code-frameworks including the SpM definitions each of which must be completed manually by the programmer writing the body of the SpM.

(2) Automatic detection of error / inconsistencies in timing specification: The inconsistency in the timing specifications of SpMs and SvMs can be detected by ViSTMO. For example, the guaranteed completion time (GCT) of an SvM should be less than the deadline set by a client for result return. However, this capability is yet to be implemented.

(3) Checking parameter type of an SvM: The parameter for any SvM must be organized as one of a
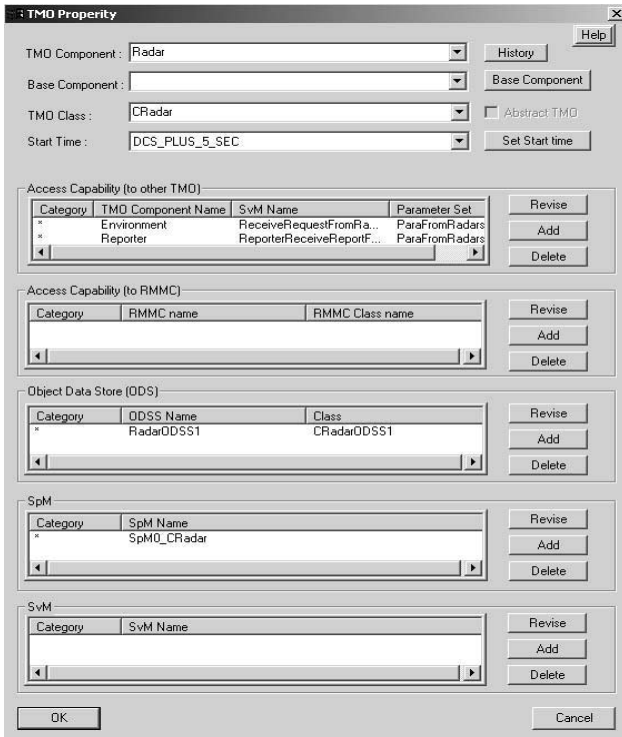
4

Figure 3. A dialog box for the creation of a TMO

"access capability (to other TMOs)" section in a dialog box for defining the properties of the client TMO shown in Figure 3. The programmer should next select the name of a server TMO and the name of a server SvM. Later when a client method, SpM or SvM, is created, the programmer should specify one or more types of service requests to be made. The service request types are discussed in the next section. These are the steps involved in creating a service-call link from a client to a server TMO with the previous version of ViSTMO.

However, a recently added feature allows the programmer to connect a method in a client TMO and a method in a server TMO by drawing a link between them. As soon as a line is drawn, a pop-up window will open up to induce the programmer to fill in a service request type. Figure 4 shows a pop-up window just after the programmer drew a line linking a client method, SpM0_Cradar in Radar, and a server method, ReceiveRequestFromRadar, in the server TMO which is Environment. Also, the programmer can verify the service-call link between two TMOs by double-clicking the line between them on the screen. Once the programmer double-clicks the line, a pop-up window almost identical to that in Figure 4, will open up showing the name of the client TMO, the name of the client method, the name of the server TMO, the name of the server method, the service request type, and the parameter structure type of the server method. Appropriate code-frameworks for service requests and result returns will
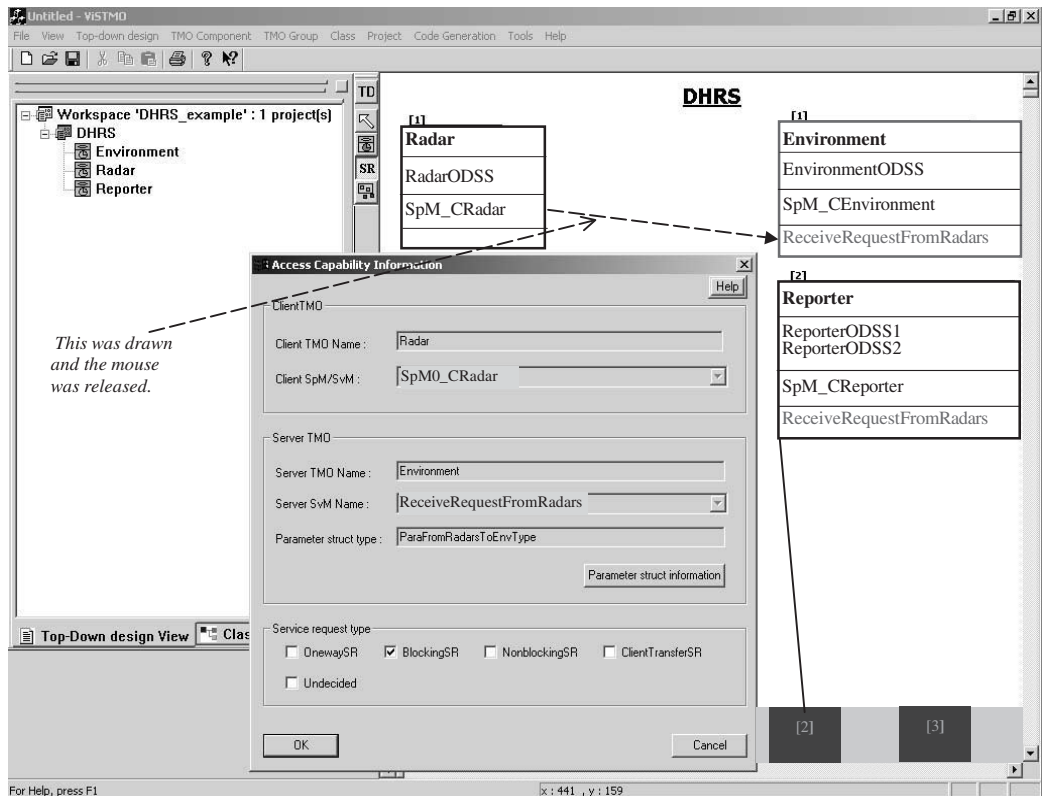
single structured type (i.e., "struct" in C++). ViSTMO forces the SvM designer to create the structured parameter type. It also forces the designer of a client to use the parameter type defined by the SvM designer in creating an actual parameter set. In this way, ViSTMO ensures type consistency of the parameter set passed between the SvM and its clients.

Three most recently realized features of the Graphics-based Design Editor are described below.

(1) Automatic generation of code-frameworks linking clients to server TMOs under the guides of drawing inputs: In order to link a client to a server TMO, a programmer should push the add button in the

Figure 4. A dialog box for linking a client to a server TMO

5

then be generated by the Code- framework Generator. Such code-frameworks will be discussed in section 4.2.

(2) TMO-to-node assignment: With this recently added feature the programmer can assign a TMO to a specific node by drawing a line linking the TMO and the icon that represents the node. For example, a line can be drawn in Figure 4 to link Reporter TMO to an icon representing node2. The information on the TMO-to-node assignment will be used by the Code-framework Generator during code generation. The Code-framework Generator will generate a separate main.cpp file for each node and a separate TMO environment configuration file named "config.ini" that is node-dependent. This config.ini file contains information such as the number of nodes in the distributed computing network, master node IP address, etc.. For more detailed information about config.ini file, please refer to the TMO Programming Manual [TMO02].

(3) RMMC-access creation: A GUI for interactive design of RMMC and connection of TMO methods to RMMCs was also a recent addition to ViSTMO. In order to create an RMMC-access class, the programmer should provide the following information.
- Name of RMMC-access class
- Name of the event message structure type and its data members
- Name(s) of state message variable(s), name of each state message structure type and its data members.

Preliminary experiments have indicated that ViSTMO can relieve the TMO programmer substantially of the burden of the manual coding.

4.2 Code-framework Generator

TMO programming is enabled by provision of a set of APIs that wrap around the execution support services of TMOSM rather than a new programming language. This API set is called the *TMO Support Library* (TMOSL). TMOSL contains C++ classes which serve as base classes during the course of defining TMOs and some of their

components, i.e., ODSSs, TMO-access gates, etc.. Therefore, the TMO programmer who does not have access to ViSTMO creates application-specific ODSS classes, TMO classes, etc., by inheriting base classes in TMOSL first.

The Code-framework Generator produces a C++ framework of each application TMO designed with the aid of the Graphics-based Design Editor. This C++ code-framework includes TMO class definitions and full details of constructors each of which includes the registration of the TMO and its methods with the execution engine. The main() function containing instantiations of TMOs, gate objects, etc., is also generated. The only thing that the TMO programmer needs to do in order to complete the implementation of a TMO is to add function bodies into this framework.

Since TMO programming is enabled by provision of TMOSL rather than a new programming language devised to support the TMO, some TMO-specific codes that are not needed in conventional OO programming and are related to defining and passing onto TMOSM relevant timing requirements, ODSS access requirements, and other parameters, cannot take strongly irredundant concise forms. On the other hand, the Graphics-based Design Editor takes essential information from the human designer in strongly irredundant concise forms. The Code-frame Generator generates a C++ TMOSL code-framework, which takes a somewhat redundant form, from the maximally concise information coming from the Graphics-based Design Editor. The TMO-specific code-frameworks contain codes for the following;

(1) ODSS registration: TMO execution engine needs to support dynamic locking and release by TMO methods of ODSSs. So it should do some

```
void main() {
  StartTMOengine();
  TMOGateClass Gate_for_Reporter_ReporterReceiveReportFromRadars ("Reporter",
            "ReporterReceiveReportFromRadars", tm4_DCS_age(DCS_PLUS_5_SEC));
  TMOGateClass Gate_for_Environment_ReceiveRequestFromRadars ("Environment",
            "ReceiveRequestFromRadars", tm4_DCS_age(DCS_PLUS_5_SEC));
  AACclass AAC1 (tm4_DCS_age(WARMUP_DELAY_SECS), tm4_DCS_age(UNTIL_DCS_PLUS_2_HOURS),
            EVERY_1_SECOND, EST_1_MILLISECOND, LST_5_MILLISECOND, DEADLINE_MSEC_SpM);
  /*********************** TMO :: Radar ***********************/
  SpM_RegistParam            Radar_SpM0_CRadar_Register_Info;
  Radar_SpM0_CRadar_Register_Info.build_regist_info_AAC(AAC1);
  CRadar   Radar ("Radar" /* TMO Name */,            DCS_PLUS_5_SEC  /* TMO Start time */,
  &Gate_for_Environment_ReceiveRequestFromRadars, &Gate_for_Reporter_ReporterReceiveReportFromRadars,
            Radar_SpM0_CRadar_Register_Info        /* SpM Register Para. for SpM0_CRadar */);
  /*********************** TMO :: Environment ***********************/
  SvM_RegistParam            Environment_ReceiveRequestFromRadars_Register_Info;
  _tcscpy (Environment_ReceiveRequestFromRadars_Register_Info.Name, "ReceiveRequestFromRadars");
  Environment_ReceiveRequestFromRadars_Register_Info.GuaranteedCompletionTime = GCT_20_MILLISEC;
  Environment_ReceiveRequestFromRadars_Register_Info.PipelineDegree = DEFAULT_SvM_Pipelinedegree_3;
  Environment_ReceiveRequestFromRadars_Register_Info.MaxInvocations =
                              DEFAULT_SvM_MIR_5_TIMES_PER_SEC;
  SpM_RegistParam   Environment_SpM0_CEnvironment_Register_Info;
  Environment_SpM0_CEnvironment_Register_Info.build_regist_info_AAC (AAC1);
  CEnvironment      Environment ("Environment" /* TMO Name */,        DCS_PLUS_5_SEC  /* TMO Start time */,
        Environment_ReceiveRequestFromRadars_Register_Info     /* SvM Register Para. */,
        Environment_SpM0_CEnvironment_Register_Info   /* SpM Register Para. for SpM0_CEnvironment */);
  MainThrSleep(); }
```

Figure 5. C++ main function generated by the Code-framework Generator

6

bookkeeping for each ODSS.

(2) TMO registration

   a. SpM related: TMO execution engine needs some information about SpM, e.g., timing specifications (AAC), ODSSs needed, etc., to do execution scheduling including concurrency control.

   b. SvM related: TMO execution engine needs some information about SvM, e.g., GCT and ODSSs needed, to do execution scheduling including concurrency control.

   c. TMO related: TMO network configuration manager needs this information to support interactions among remote TMOs.

4.2.1 Generation of a main function. Figure 5 shows a sample C++ main function generated by the Code-framework Generator. In this example, the main function starts TMOSM by calling the TMOSL API, StartTMOengine(). Then two gate objects are constructed and the constructor of each gate tells the execution engine to establish an efficient path to the remote SvM, keep the path inaccessible until the time, Service_Start_Time, arrives, and then make the path usable. A client TMO can use these gate objects to access ReporterReceiveReportFromRadars SvM in Report TMO and ReceiveRequestFromRadars SvM in Environment TMO, respectively. An example of use of a gate object will be discussed in the next section. Next, an AAC object is constructed by using the specification of when a SpM should be activated for its one-shot or periodic execution, when it should be deactivated, the interval between periodic execution-starts, earliest start-time in each execution round, and latest start-time in each executions round, and GCT of each round.

Finally, two TMOs, Radar and Environment, are constructed by using the specification of TMO name, TMO start-time, gate objects that will be used (Environment TMO does not use any gate object), and timing specifications for two SpMs and an SvM (Radar TMO does not have any SvM). In the case of the timing specification of an SvM, the programmer should specify the name of the SvM, a GCT, a pipelining degree (i.e., the number of invocations of the same SvM that can progress concurrently), and a maximum invocation rate. Direct manual coding of the statements in Figure 5 would be a lot more burdensome to the TMO network programmer than obtaining the main function through interactive use of ViSTMO.

4.2.2 Generation of the code-framework for a TMO class definition. Figure 6 shows the constructor of CRadar class and the framework of the body of the SpM of CRadar class, SpM0_CRadar generated by the the Code-framework Generator. In this example, the generated code-framework for SpM0_CRadar contains declarations of parameters and prototypes of service request calls through the gates for the other two TMOs, Environment TMO and Reporter TMO. These codes, which were generated by the Code-framework Generator based on the information provided by a programmer through the Graphics-based Design Editor, form an incomplete body of the SpM.

It shows that selection of a service request (SR) type after drawing a link from Radar TMO to Environment TMO and similar selection for a link to Reporter TMO during the graphics-based design editing results later in the generation of (1) C++ codes for saving pointers to relevant gate objects as data members of CRadar, (2) the codes for declaration of relevant parameters, and (3) the prototype codes for SR call of the selected type. In SpM0_CRadar in Figure 6, gate0 object is used to make a non-blocking SR to ReceiveRequestFromRadars SvM of Environment TMO. The first parameter of this SR is a reference to a data structure to be passed to the SvM, and the second one is the size of this data structure. The third one will carry a time-stamp to be set by the execution engine running this client upon completing the execution of the non-blocking SR. The client TMO can use this time-stamp to retrieve the service results coming from the SvM. Gate objects provide flexible interfaces for users to make various kinds of SRs to server TMOs, such as non-blocking SR, blocking SR, non-blocking result retrieval, and blocking result retrieval, etc..

Neither a C++ compiler nor TMOSL can access the source codes of all the TMOs engaged in cooperative distributed computing. Therefore, there are no mechanisms in TMOSM or TMOSL for checking the type consistency. Since ViSTMO is designed to access the essential designs of all the TMOs engaged in a distributed computing application, it has a wider vision than a conventional C++ compiler does. Therefore, it can check

```
#include    "CRadar.h"
void CRadar::SpM0_CRadar(MicroSec) {
  tmsp      TimeStamp0; ParaFromRadarsToEnvType  para_ParaFromRadarsToEnvType;
  ParaFromRadarsToReporterType  para_ParaFromRadarsToReporterType;
  // TO DO : Add your implementation here
  gate0->NonBlockingSR (&para_ParaFromRadarsToEnvType,
                  sizeof(para_ParaFromRadarsToEnvType),  TimeStamp0);
  gate1->OnewaySR (&para_ParaFromRadarsToReporterType,
                  sizeof(para_ParaFromRadarsToReporterType)); };
CRadar::CRadar(char * TMO_name, tms TMO_start_time, TMOGateClass * para_gate0,
      TMOGateClass * para_gate1, SpM_RegistParam SpM_RegistParam_of_SpM0_CRadar)
{ gate0 = para_gate0;                gate1 = para_gate1;
  /* Declare an ODSS to be in the group of ODSSs to be accessed by SvM */
  SpM_RegistParam_of_SpM0_CRadar.build_regist_info_ODSS (RadarODSS1.get_id(), RW);
  /* Preparation of the registration parameters related to SpM */
  if (!Register_SpM((PFSpMBody)SpM0_CRadar, &SpM_RegistParam_of_SpM0_CRadar))
          TMOSLprintf("SpM register failed :: SpM0_CRadar\n");
  /* TMO Registration */
  TMO_RegistParam  TMO_RegistInfo;    _tcscpy(TMO_RegistInfo.Name, TMO_name);
  TMO_RegistInfo.StartTime = TMO_start_time;
  if (!Register_TMO(&TMO_RegistInfo))
          TMOSLprintf("TMO register failed :: %s \n", TMO_name); };
```

Figure 6. A sample *.cpp file generated by the Code-framework Generator

7

www.manar

the type consistency between the formal parameters defined as a part of an SvM and the actual parameters created in a client TMO. In the case of Figure 6, ViSTMO can ensure that a client TMO makes a SR with a parameter of type ParaFromRadarsToEnvType, which was created as a part of the definition of the server SvM, ReceiveRequestFromRadars. Also, in both Figure 5 and Figure 6, the words in bold-face are input by the programmer only once through the Graphics-based Design Editor. These examples show how ViSTMO can increase the programming efficiency and the system reliability.

6. Discussion and Conclusion

Fully manual coding of TMOs involves quite a bit of repeated routine work. ViSTMO provides the user-friendly Graphics-based Design Editor with which programmers can input essential information in concise and efficient manners. It helps minimizing the number of errors that can be made by the programmers and eliminating or minimizing the amount of redundant data that need to be provided by the programmers. The design methodology supported is of a top-down stepwise refinement type discussed in [Kim97]. Since graphics-based representations of TMO network designs are much easier to understand than most of the purely character-based representations, ViSTMO also greatly enhances maintainability of application TMO networks.

An important advantage of ViSTMO is that it minimizes the gap between design and programming. Most modeling tools devised to help distributed programmers are based on highly abstract modeling approaches [Ros02]. On the other hand, ViSTMO supports smooth transitions from designs to executable codes by generating not only class frameworks but also prototype codes for remote method calls, etc. ViSTMO can be extended to support design and implementation of networks of TMOs running on multiple types of platforms.

Currently, several versions of TMO execution engines, such as TMOSM/XP (TMOSM layered on Windows XP and Windows 2000) and TMOSM/CE (TMOSM layered on Windows CE), are available. TMOSM can also run over different communication mechanisms, such as SOCKET, CORBA ORBs (Object Request Brokers), and DCOM facilities. There are slight but inevitable differences among the middleware layered on different platforms and communication protocols. ViSTMO can play a significant role in eliminating most differences among various TMO execution engines from the concerns of TMO programmers. Some limited experiments with ViSTMO in RT application design and implementation have been conducted with encouraging results.

References

[Kim95] Kim, K.H., Mori, K., and Nakanishi, H., "Realization of Autonomous Decentralized Computing with the RTO.k object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme", *Proc. ISADS '95 (IEEE Computer Society's 2nd Int'l Symp. on Autonomous Decentralized Systems)*, Phoenix, AZ, April 1995, pp.305-312.
[Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", IEEE *Computer*, August 1997, pp.62-70.
[Kim99] Kim, K.H., Ishida, M., and Liu, J., "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. ISORC '99 (IEEE CS 2nd Int'l Symp. on OOReal-time distributed Computing)*, May 1999, pp.54-63.
[Kim00] Kim, K.H., "APIs for Real-Time Distributed Object Programming", IEEE *Computer*, June 2000, pp.72-80.
[Kim02] Kim, K.H., "Commanding and Reactive Control of Peripherals in the TMO Programming Scheme", *Proc. ISORC 2002 (IEEE CS 5th Int'l Symp. on OO Real-time distributed Computing)*, Washington, D.C., April 2002, p. 448-456.
[Kop97] Kopetz, H., '*Real-Time Systems: Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, ISBN: 0-7923-9894-7, Boston, 1997.
[Mol01] Molina, P.J., Pastor, O., Marti, S., Fons, J.J., and Insfram, E. "Specifying conceptual interface patterns in an object-oriented method with automatic code generation", *Proc. User Interfaces to Data Intensive Systems (UIDIS), 2001*, Zurich, Switzerland, May 2001, pp72-79.
[Mor82] Mori, K. and Ihara, H., "Autonomous Decentralized Loop Network", *Proc. IEEE CS COMPCON*, Spring 1982, pp. 192-195
[Mor86] Mori, K., et. al., "Autonomous Decentralized Software Structure and Its Application.", *Proc. Fall Joint Computer Conference*, Dallas, Texas, November 1986, pp. 1056-1063.
[Nie00] Niemann, M., and Bardohl, R., "Tool-Based Specification of Visual Languages and Graphical Editors", *Proc. 6th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Berlin, Germany, March 2000.
[OMG01] Object Management Group, "UML Profile for Schedulability, Performance, and Time revised submission", OMG Document No. ad/01-06-14, June 2001.
[OMG02] Object Management Group, "Chapter 24. Real-time CORBA", in *CORBA Specification*, Version 2.6.1, http://www.omg.org/technology/documents/formal/corba_2.htm, May, 2002.
[Pro99] Prosise, Jeff, '*Programming Windows with MFC*', 2nd Edition, Microsoft Press, Redmond, 1999.
[Ros02] Rational Software Corporation, http://www.rational.com/
[Sch00] Douglas C. Schmidt and Fred Kuhns, "An Overview of the Real-time CORBA Specification", IEEE *Computer*, June 2000, pp. 56-63.
[Sel00] Selic, B., "A Generic Framework for Modeling Resources with UML", IEEE *Computer*, June 2000, pp. 64-69.
[Sof] Softwire Technology, "Softwire - Graphical Programming for Visual Studio .Net", white paper in http://www.softwire.com/media/pdfs/white-1.pdf .
[TMO02] DREAM Lab, University of California , Irvine, "TMO Programming Manual", http://dream.eng.uci.edu/ece147/serious.htm.