Model-driven Development of Complex Routing Protocols with SDL-MDD

Alexander Geraldy, Reinhard Gotzhein, and Christoph Heidinger

Networked Systems Group, Department of Computer Sciences,
University of Kaiserslautern, Germany
{geraldy, gotzhein, c_heidin}@cs.uni-kl.de

Abstract. In this paper, we provide a substantial case study of developing complex routing protocols, applying SDL-MDD, a model-driven development process with SDL as modeling language. After a survey of SDL-MDD, we present a generic routing architecture, which supports several forms of routing protocol composition. This architecture is then instantiated first with *ReBaC2*, a novel repair-based clustering algorithm for ad-hoc networks, and then extended by *AodvLight*, a reactive routing protocol for route discovery across clusters. The specified SDL models are the basis for model-driven implementation and simulation.

1 Introduction

Model-driven development (MDD) [1] is a software engineering approach where the formal system model guides and directs all development activities, ranging from system design over code generation and deployment to system maintenance, resulting in quality improvements and productivity gains. In previous work [12], we have introduced SDL-MDD, a model-centric, domain-specific development process based on SDL [10], ITU's Specification and Description Language. In the early phases, SDL-MDD benefits from structuring and reuse methods, for instance, SDL design patterns and micro protocols. A particular strength of SDL-MDD is the availability of a semantically integrated tool suite that covers all aspects of model-driven development with SDL including model-driven code generation, automatic environment interfacing, and performance simulation.

In this paper, we report on the application of SDL-MDD to the development of a complex routing protocol for mobile ad-hoc networks. These networks are formed spontaneously by nodes within reach that have matching wireless communication facilities. As there is no fixed infrastructure, these networks are highly dynamic regarding their topology, which poses big challenges to the development of suitable communication protocols in general, and to routing protocols in particular. To cope with these difficulties, we have devised a generic clustering algorithm for ad-hoc networks called Repair-based Clustering (ReBaC2), which establishes disjoint sets of nodes controlled by a metric that can be tuned to specific network situations. This clustering is then the basis for a hierarchical routing scheme, using a tree-like structure within clusters and a simplified version of Ad-hoc On-demand Distance Vector routing called AodvLight for reactive route discovery across clusters. To cope with system complexity, we start by specifying micro protocols [7], which are then composed by instantiating a generic routing architecture [2]. Results show that SDL-MDD with its tool suite, the micro-protocol approach, and the generic routing architecture form a powerful basis that supports all aspects of developing complex routing protocols.

The rest of this paper is structured as follows: In Sections 2 and 3, we survey SDL-MDD and our generic routing architecture, respectively. This architecture is instantiated in

Section 4, where Repair-based Clustering (ReBaC2) is introduced. In Section 5, ReBa2C is composed with AodvLight, yielding a hierarchical routing algorithm. Simulation results are presented in Section 6, and conclusions are drawn in Section 7.

2 Model-driven Development with SDL

SDL-MDD [12] (see Fig. 1) is a model-driven development process [1] that is based on and extends OMG's MDA [13], with SDL [10] as modeling language. The computation-independent model (CIM) is expressed by message scenarios, specified with MSC [9], and informal text. For the formal specification of the platform-independent and platform-specific models (PIM and PSM), we use SDL [10], ITU's Specification and Design Language for distributed systems and communication protocols. The process steps from CIM over PIM to PSM are supported by several structuring and reuse methods, in particular transformation heuristics [12], SDL design patterns [6], SDL components [3], and micro protocols [7].

SDL components [3] are ready-to-use, self-contained design solutions, supporting both structuring of complex systems and reuse of well-proven solutions. In [7], we have extended this concept to distributed components providing a single functionality, called *micro protocols*. A micro protocol is instantiated by creating a set of protocol entities that interact to provide a distributed functionality. By composing micro protocols, more complex protocols and protocol stacks can be obtained.

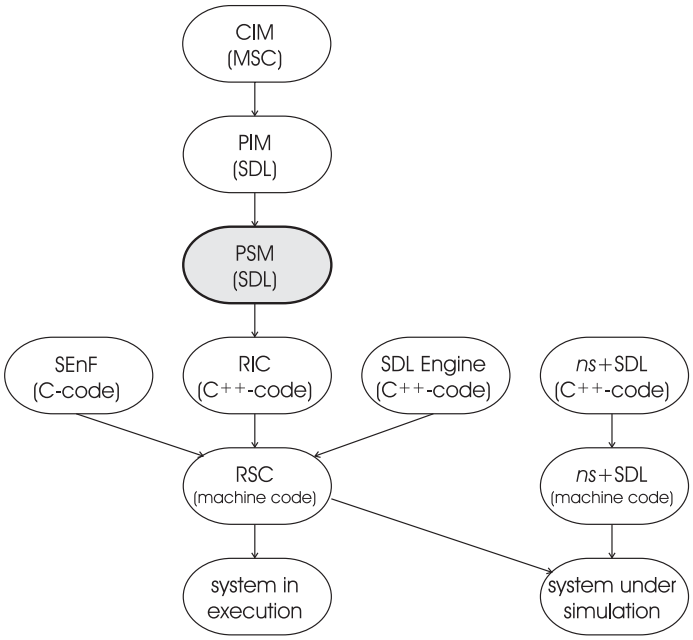

Figure 1: SDL-MDD – Model-driven development with SDL

The platform-specific SDL model (PSM) is the starting point for model-driven implementation. From SDL specifications, it is possible to generate code in two steps. In the first step, intermediate code in languages such as C or C++ is compiled. This code can be executed

in different runtime-environments and is therefore referred to as *Runtime-Independent Code* (*RIC*) in Fig. 1. To generate intermediate code, commercial tools provided, for instance, by PragmaDev and Telelogic, are available. In our work, we have developed the SDL-to-C++ transpiler ConTraST [5], which in addition automatically generates the documentation of micro protocols [4].

To be executed on a specific target system, the RIC is compiled to machine code referred to as *Runtime-Specific Code* (*RSC*) (see Fig. 1), using a platform-specific compiler. To execute the RSC, an SDL engine for the target system is required in addition. This SDL engine comprises all functionality to initialize and execute the SDL system, e.g., to build up the system structure, to select, schedule, and execute fireable transitions, and to transfer signals between SDL processes.

To implement open SDL systems, i.e. systems interacting with their environment, one or more interfaces satisfying the semantics of the SDL signaling mechanism, in particular, the type of interaction and the interaction formats, are needed. For this purpose, we have developed a generic, specification-independent library of interfacing routines, called *SDL Environment Framework* (*SEnF*). Based on configuration information supplied by the SDL compiler, interfacing routines for different combinations of operating systems, communication technologies, and IO devices are automatically determined and added to the generated code.

In our work [11], we have shown that the platform-specific SDL model can also be used as starting point for model-driven performance simulations. In Fig. 1, the corresponding process steps are shown. As in case of implementing SDL systems, runtime-specific code (RSC in Fig. 1) is generated. For functional and performance simulations, this code is then executed under the control of a simulator. A simulator that can run code generated from SDL models is *ns*+SDL [11]. The advantage of generating both production and simulation code from the same SDL model with the same compiler is that performance simulations more faithfully reflect the performance of the deployed system.

3 A generic routing protocol architecture

In computer networks, different routing algorithms may be needed for different purposes. For instance, specialized routing algorithms may be used for unicast, multicast, and broadcast communication. Also, flat/hierarchical, global/aggregated, and proactive/reactive algorithms may be applied, depending on network size and frequency of route requests. The choice of routing algorithms is further constrained by node mobility and corresponding topology changes.

To satisfy these diverse needs, we have devised a generic routing protocol architecture (see also [2]), which provides a design framework for the development of specialized, complex routing protocols. Fig. 2 shows the abstract architecture, consisting of network layer and adjacent protocol layers, with the network layer being refined into components *PacketForwarder*, *RouteDiscoverer*, and *DeMux*. For interaction between and within layers, several channels and messages sent along these channels are specified. The basic routing functionality of the network layer is as follows:

- *PacketForwarder* accepts messages from *TransportLayer* (SendData). Apart from the data to be sent, *TransportLayer* passes the destination address, from which the type of communication (unicast, multicast, broadcast) is derived.

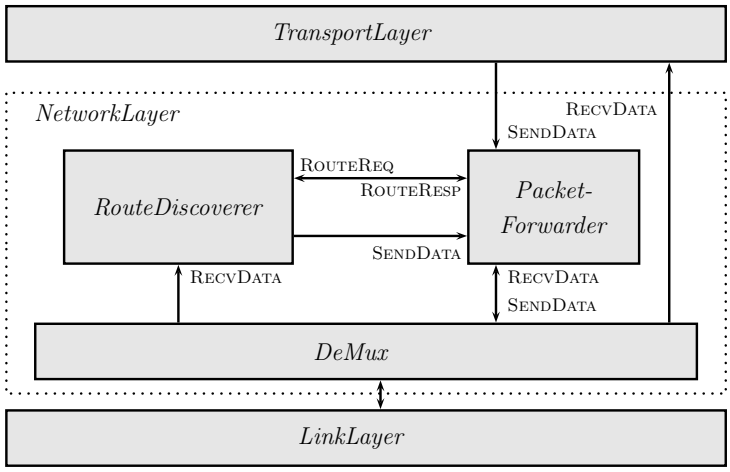

Figure 2: Generic routing protocol architecture (level 1)

- Upon acceptance of a message from *TransportLayer*, *PacketForwarder* consults with *RouteDiscoverer* to determine a set of feasible routes (RouteReq). When this set (RouteResp, consisting, e.g., of complete routes or next hop addresses) is returned, the message is sent via *DeMux*.
- When a packet is received from *LinkLayer* (RecvData), *DeMux* determines the local component to process the packet. If the packet has reached its final destination node, it is forwarded to *TransportLayer*. If it is to be sent to another node, it is given to *PacketForwarder*.
- Upon reception of a packet from *DeMux*, *PacketForwarder* checks whether the packet carries sufficient routing information. If this is the case (e.g., if source routing is performed), it processes the packet and sends it via *DeMux*. Otherwise, it first consults with *RouteDiscoverer* to determine a feasible set of routes (see above).

In order to incorporate the collection of network state information, the basic routing functionality is extended:

- *RouteDiscoverer* sends management packets in order to acquire information about the network state. To reduce interdependencies of routing protocols, these packets are sent via *PacketForwarder*, where they are treated like all other packets. This means that if the packet carries sufficient routing information, it is sent via *DeMux*. Otherwise, *PacketForwarder* consults with *RouteDiscoverer* to determine a set of feasible routes before sending the packet.
- The functionality of *DeMux* is extended such that arriving packets with destination *RouteDiscoverer* are delivered accordingly.

Please note that this generic routing protocol architecture is extremely flexible, it may host, for instance, proactive/reactive, flat/hierarchical, source/distributed, and unicast/multicast/broadcast route discovery algorithms. For this purpose, *RouteDiscoverer* is decomposed as shown in Fig. 3. Here, R_1, R_2, ... R_n denote components realizing a specific route discovery algorithm, which are *composed in parallel*. Two glue components are added:

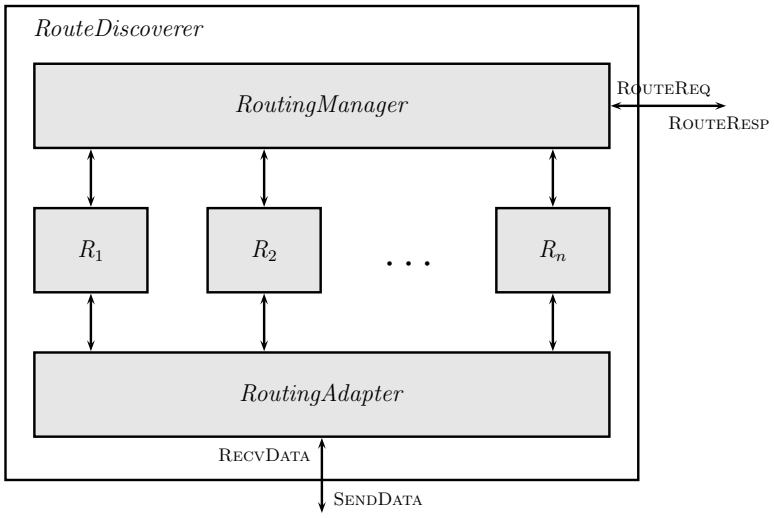

Figure 3: Generic routing protocol architecture (levels 2..n): *RouteDiscoverer*

- *RoutingManager* interacts with *PacketForwarder*, receiving RouteReq and returning RouteResp signals. Based on the type of route request (e.g., unicast, multicast) and the available routing algorithms, one or more route discovery components are selected and scheduled.
- *RoutingAdapter* encapsulates outgoing route management packets, and demultiplexes incoming packets containing network state information.

The architecture shown in Fig. 3 can also be used to refine R_1, R_2, ... R_n, yielding a *hierarchical composition* of route discovery algorithms.

4 ReBaC2: A novel dynamic clustering approach

In this section, we apply SDL-MDD to the design of Repair-based Clustering (ReBaC2), a novel dynamic clustering approach for ad-hoc networks, instantiating the generic routing architecture presented in Section 3. Mobile ad hoc networks with large numbers of nodes and high node density entail considerable topological complexity. If flat and proactive routing schemes are used, this leads to large routing tables, long status messages, and thus to very high routing overhead. Approaches to reduce routing overhead are to build up hierarchies to establish abstract views of the real network topology, and to apply reactive routing schemes. Dynamic clustering is an approach to set up a hierarchical structure in a network.

Clustering is a self-organizing process during which network nodes group themselves into logical units called *clusters*. The whole network is subdivided into such clusters, which may, but need not be disjoint. A cluster is usually a set of nodes sharing common parameters. For example, the nodes of a cluster may be topologically close to each other. Being topologically close, the nodes of a common cluster can communicate more easily with each other than with arbitrarily chosen nodes of the complete network.

ReBaC2 was developed with several special design goals in mind: Firstly, there shall be no separation between the setup phase and the maintenance phase, which could have different

rules and behavior. In other words, there is only one phase, called *repair phase*, with one unique set of clustering rules. This has the advantage that when the network topology is changed due to new nodes or node movements, the logical structure needs to be repaired only, but not be set up from scratch. Secondly, the clustering protocol shall have a modular metric, which can be defined separately from the clustering rules, and can therefore be replaced easily. Thus, the clustering protocol can be adapted to specific network situations. Thirdly, while clusters are formed, status information for routing among cluster members shall be collected proactively.

With ReBaC2 [8], a cluster is a set of network nodes that are close to each other according to a given metric, with a distinguished node called *cluster head*. Cluster nodes except the cluster head are called *cluster members*. All clusters in the network are disjoint, i.e. at every time each network node belongs to exactly one cluster. The nodes of a cluster form a logical tree rooted at the cluster head. Some of the cluster nodes may be *gateway nodes*, i.e. nodes offering a direct link to gateway nodes of other clusters.

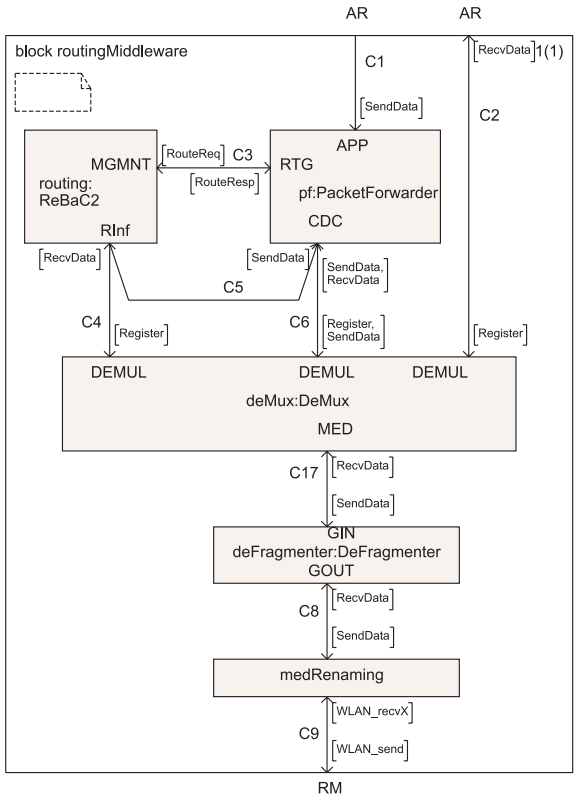

Figure 4: Routing middleware

For routing between nodes belonging to the same cluster, the logical tree structure established during cluster formation is exploited. Here, the cluster head has global status information, i.e. it knows all cluster members and paths along the logical tree. Cluster members have limited status information, knowing their cluster head and the next hop towards it

only. In addition, cluster heads know the cluster heads of neighboring clusters, and gateway nodes to these clusters. This knowledge forms the basis for the discovery of routes between nodes that belong to different clusters.

In summary, apart from being a clustering approach, ReBaC2 classifies as a proactive routing protocol. Consequently, we have instantiated the generic routing architecture presented in Section 3, using SDL as design language. The top level of the concrete architecture is shown in Fig. 4, where the SDL block ReBaC2 replaces the generic component *RouteDiscoverer*. Furthermore, components for fragmentation and defragmentation, and for renaming have been added, to cope with the constraints of real wireless LANs.

In Fig. 5, the block ReBaC2 representing the generic component *RouteDiscoverer* is refined, instantiating level 2 of the generic routing architecture. More specifically, the generic glue components *RoutingManager* and *RoutingAdapter* (see Fig. 3) are instantiated by specific ReBaC2 protocol components. These glue components encapsulate the actual protocol behavior of the blocks AliveSender and AliveReceiver, which handle the periodic data exchange of ReBaC2, and of the block Control that coordinates this data exchange.

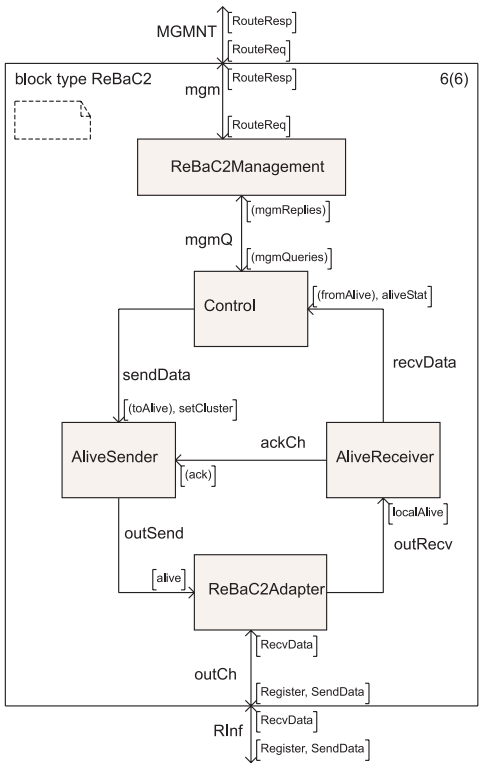

Figure 5: Block type ReBaC2

When ReBaC2 is started in a network node, it initializes this node as the head of a cluster without any members. This immediately leads to a consistent yet probably suboptimal state of the clustering structure. After this, the so-called repair phase continuously attempts to improve the clustering, e.g. by joining another cluster if this increases the metric value of the

own cluster. For this purpose, the node sends a join-request to neighboring clusters, which determine if the join would violate the local metric constraints, e.g. if the resulting cluster would show too many hops or too many member nodes. Neighboring clusters that do not detect any violation respond by sending a join-accept signal, offering membership. The head of the cluster requesting membership then decides which neighboring cluster to join.

For the repair decision of the own cluster and for the admittance test of neighboring clusters, the same set of mathematical rules is used. We have extracted these rules into an SDL data type, which may be replaced to change the desired cluster sizes and clustering metric, e.g. hop count, member count, link quality, and battery power. Therefore, ReBaC2 can be adapted to satisfy special requirements, and is therefore a generic clustering approach. More details on ReBaC2 can be found in [8].

5 ReBaC2/AodvLight

In this section, we extend the routing middleware presented in Section 4 by a protocol for reactive route discovery between nodes that belong to different clusters. This protocol exploits the abstract view and gateway links established by ReBaC2 without actually being aware of its existence, which is clearly in line with our design goal of defining self-contained micro protocols and composing them into complex protocols by instantiating our generic routing protocol framework.

More specifically, we extend ReBaC2 with simplified version of Ad-hoc On-Demand Distance Vector routing [14] called *AodvLight* [2]. In AodvLight, unidirectional routes are discovered on demand, by flooding a RREQ message starting with the source node. As soon as a node - possibly the destination node - knows a feasible path, it returns a unicast RREP message, which takes the backward path recorded in the intermediate nodes to inform the source node.

When composed with ReBaC2, AodvLight operates on the upper hiearchy, exploiting the overlay network structure formed by all cluster heads. To discover routes between nodes belonging to different clusters, only this overlay network is searched. For this purpose, RREQ messages are flooded among all cluster heads. Since cluster heads have global status information of their cluster, they know whether the destination node belongs to their cluster. If so, a route has been established, and a RREP message is returned. If not, the RREQ message is forwarded to the cluster heads of all neighboring clusters.

It should be pointed out that flooding a message among cluster heads requires that unicast routing between cluster nodes is performed, using the logical tree structure of clusters and the gateways to neighboring clusters. Thus, flooding a RREQ message in fact means sending unicast messages to the heads of neighboring clusters on unicast paths that have already been established proactively. Again, we emphasize that this is accomplished by separating concerns in our generic routing framework, and not by modifying AodvLight before composing it with ReBaC2.

Adding AodvLight to the routing middleware of Section 4 is achieved by reinstantiating the generic routing architecture in Figs. 2 and 3. Here, *RouteDiscoverer* is now instantiated by the SDL block ReBaC2Aodv (see Fig. 6), which is in turn decomposed into glue components ReBaC2AodvManager and ReBaC2AodvAdapter, encapsulating the routing protocols ReBaC2 and AodvLight.

Fig. 7 shows an excerpt of the SDL specification of the ReBaC2AodvManager. For each route request asking for a set of destination nodes, a manager instance is created. After the start transition, the incoming RouteReq is answered in three steps:

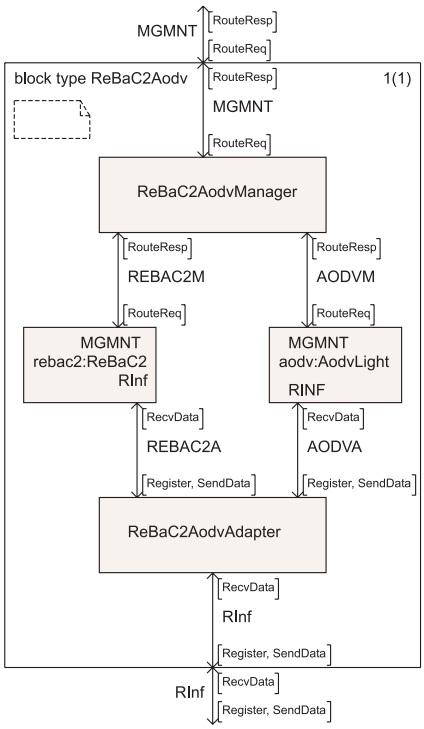

Figure 6: Block type ReBaC2Aodv

1. The RouteReq is forwarded to ReBaC2. ReBaC2 uses its local knowledge of the cluster to resolve some of the destination nodes, as far as they are cluster members or neighboring cluster heads.
2. When ReBaC2 answers with a RouteResp, some or all of the destination nodes may still be unresolved. Therefore, the partly handled request is forwarded to AodvLight.
3. AodvLight will send back a RouteResp, which potentially contains next hop nodes that are only next hop in the AodvLight overlay network. All next hop nodes are marked as unresolved and are checked and resolved using the local knowledge of ReBaC2 again.
4. The resulting RouteResp of ReBaC2 is returned to the invoking process, and the ReBaC2AodvManager terminates.

It should be pointed out that in order to compose ReBaC2 and AodvLight, no modifications of their SDL designs had to be made. This provides evidence that the previous designs were indeed self-contained and conceptually sound. It also shows that the generic routing architecture in Section 3 is a suitable basis for the development of complex routing protocols.

6 Simulation

In Fig. 8, results of a model-driven simulation run are shown. The system under simulation has been obtained by compiling the SDL model with the SDL-to-C++ transpiler ConTraST [5], yielding the RIC (see Fig. 1). The RIC is then compiled into runtime-specific machine

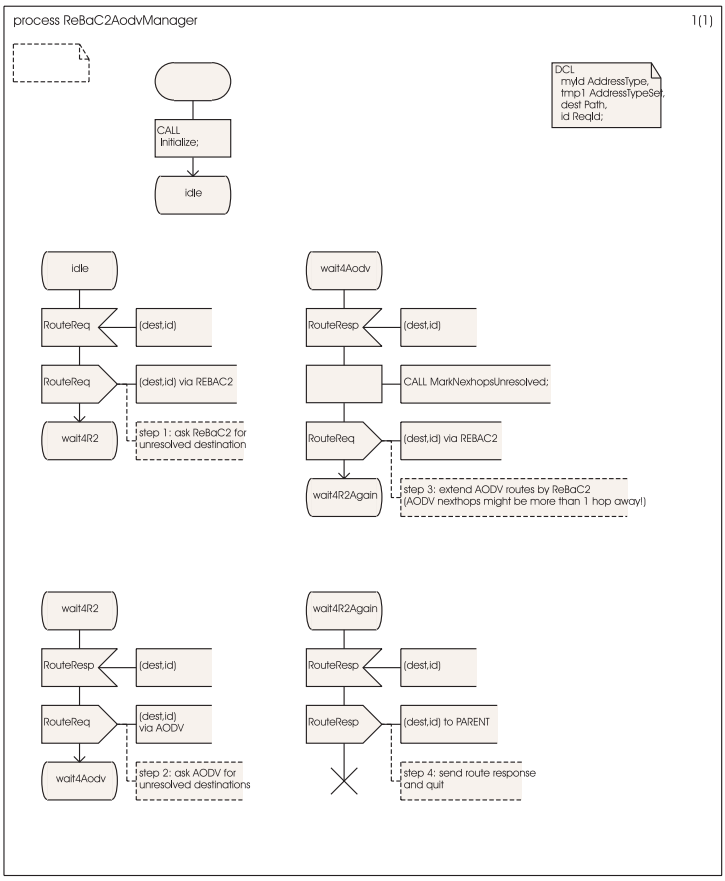

Figure 7: SDL Process ReBaC2AodvManager

code, which is augmented by an SDL Engine and environment interfacing routines. This code is then executed under the control of *ns*+SDL [11], our network simulator for SDL systems.

In the simulation, 56 nodes have been randomly placed on an area of 400x400 metres, transmission range is 73 meters. ReBaC2 has been configured to a minimum cluster size of 4 nodes, including the cluster head. In the figure, network nodes are depicted as plus signs, cluster heads as red squares, and the cluster structure is drawn with black solid lines. In total, ReBaC2 forms 8 clusters, with cluster sizes ranging from 3 to 13, and path lengths from leaf nodes to cluster heads of up to 5 hops. The reason why the small cluster has a size below the configured minimum size of 4 nodes is that the cluster head is not in direct range of nodes belonging to another cluster, and can therefore not perform a join.

In addition to the cluster structure, ReBaC2 determines gateway nodes to other clusters. Green dotted lines indicate links between gateways. This overlay structure is used by Aodv-Light to determine end-to-end paths between nodes belonging to different clusters. One such example is the overlay path given by the sequence of orange arrows. From the source node, packets travel to its cluster head, along further cluster heads towards the cluster head of the

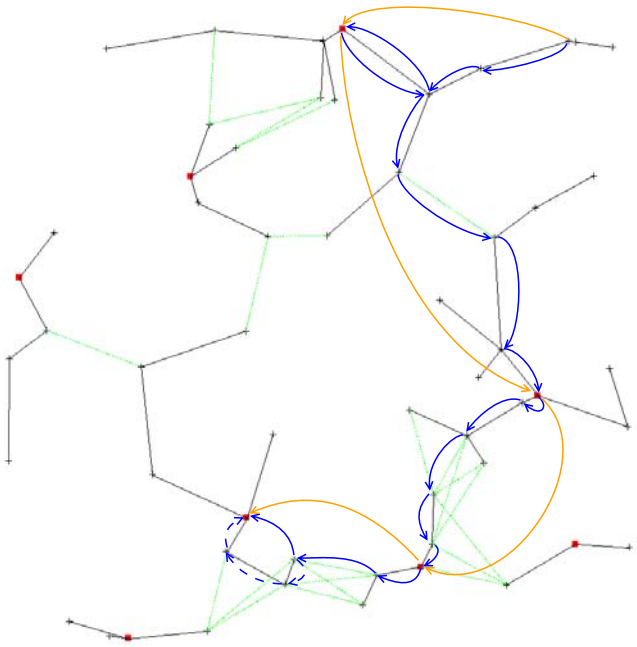

Figure 8: Simulation results of ReBaC2/AodvLight

destination, and finally to the destination. The low level links of this path are depicted as blue arrows.

Because of the overlay structure, ReBaC2Aodv does not always find shortest paths, but sometimes returns detours, e.g. as depicted by the dashed blue line in Fig. 8. At this point, optimizations should be taken into account to shorten the paths from source to destination.

7 Conclusions

In this paper, we have shown how to develop complex routing protocols for ad-hoc networks using SDL-MDD, a model-driven development process with SDL as modeling language. We have presented a generic routing framework, which supports composition and reuse of elementary routing protocols specified in SDL, and their model-driven development. Then, we have applied SDL-MDD and the generic routing framework to the micro protocol based development of a new clustering approach for mobile wireless ad hoc networks, called *ReBaC2*. ReBaC2 benefits from its single phase and single metrics solution and offers possibilites for adaptation to specific application requirements. We have further shown how to compose ReBaC2 with the reactive routing protocol *AodvLight* to get a sophisticated hierarchical routing protocol for mobile ad-hoc networks. Results of model-driven simulations provide evidence for the functional correctness of the design model.

The experience of this major case study shows that SDL-MDD combined with micro protocol based design and a generic routing framework is fully capable of handling the development of complex routing protocols, thereby mastering system complexity and saving development time. Keys to this success have been the identification of self-contained routing protocols and their composition, and the support by the integrated tool chain of SDL-MDD.

By reuse and composition, we have been able to avoid many of the usual design errors of protocol development, and have gained early feedback. Additionally, the SDL-MDD process and tools support the automatic generation of simulation and production code. Since code generation relies on the same SDL model, it can be expected that the simulation faithfully reflects the behaviour of the system in execution.

Of particular value has been the strict application of the generic routing architecture, which forces the developer to adopt the important engineering principle of separation of concerns. As a reward, we have obtained a highly modular design of a complex system and conceptual clarity, fostering maintainability and extensibility.

A key to the instantiation of the generic routing framework has been the definition of a general address type that can be used by various routing algorithms. This type has to support different addressing schemes (e.g., unicast, multicast, broadcast, n-hop cast) and routing schemes (e.g., next hop routing, source routing. Given the data type definition facilities of SDL, which are not very "'user-friendly"', it took us quite a while to find an adequate representation.

Our clustering protocol ReBaC2 offers some potential for improvements. The possibility to identify and to split existing suboptimal clusters has yet to be studied. By splitting existing clusters and joining their parts with other clusters, the overall quality of the clustering could be improved. Also, the logic tree structure of clusters can be improved to support shortest paths between cluster members and cluster head. Finally, the collection of cluster metrics can be extended to offer further possible applications of the clustering protocol. Once these improvements have been made, extensive performance simulations to compare ReBaC2 with other clustering protocols can be performed.

References

1. M. Book, S. Beyeda, and V. Gruhn. *Model-driven Software Development*. Springer, 2005.
2. I. Fliege, A. Geraldy, and R. Gotzhein. Micro protocol based design of routing protocols for ad-hoc networks. In *7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), Marrakesh, Morocco*, June 4-8, 2007.
3. I. Fliege, A. Geraldy, R. Gotzhein, T. Kuhn, and C. Webel. Developing safety-critical real-time systems with sdl design patterns and components. *Computer Networks*, 49(5):689–706, 2005.
4. I. Fliege and R. Gotzhein. Automated generation of micro protocol descriptions from sdl design specifications. In R. Reed E. Gaudin, E. Najm, editor, *SDL 2007: Design for Dependable Systems*, volume 4745 of *LNCS*, pages 150–165. Springer, 2007.
5. I. Fliege, R. Grammes, and C. Weber. ConTraST - a configurable sdl transpiler and runtime environment. In R. Gotzhein and R. Reed, editors, *System Analysis and Modeling: Language Profiles*, volume 4320 of *LNCS*, pages 216–228. Springer, 2006.
6. R. Gotzhein. Consolidating and applying the sdl-pattern approach: A detailed case study. *Information and Software Technology*, 45:727–741, 2003.
7. R. Gotzhein, F. Khendek, and P. Schaible. Micro protocol design - the snmp case study. In E. Sherratt, editor, *Telecommunications and Beyond - The Broader Applicability of SDL and MSC*, volume 2599 of *LNCS*, pages 61–73. Springer, 2003.
8. C. Heidinger. Cluster based routing in mobile ad hoc networks. Master's thesis, Networked Systems group, University of Kaiserslautern, Germany, 2008.
9. ITU-T. *Message Sequence Charts (MSC)*. ITU-T Recommendation Z.120. International Telecommunications Union, 2001.
10. ITU-T. *Specification and Description Language(SDL)*. ITU-T Recommendation Z.100. International Telecommunications Union, August 2002.
11. T. Kuhn, A. Geraldy, R. Gotzhein, and F. Rothlï¿½nder. ns+sdl - the network simulator for sdl system. In A. Prinz, R. Reed, and J. Reed, editors, *SDL 2005: Model Driven*, volume 3530 of *LNCS*, pages 103–116. Springer, 2005.

12. T. Kuhn, R. Gotzhein, and C. Webel. Model-driven development with sdl - process, tools, and experiences. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML 2006), Genua, Italy*, Oct 1-6, 2006.
13. J. Miller and J. Mukerji. *MDA Guide Version 1.0.1.* Object Management Group (OMG), 2003.
14. C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99), New Orleans, USA*, pages 90–100, 1999.