# Evaluation of UML-RT and Papyrus-RT for Modelling Self-Adaptive Systems

Nafiseh Kahani, Nicolas Hili, James R. Cordy, Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: {kahani, hili, cordy, dingel}@cs.queensu.ca

***Abstract*—This paper is an evaluation of UML for Real-Time (UML-RT) for modelling Self-Adaptive Software (SAS) systems. Using a systematic review of the different features of UML-RT (optional capsules, SAP/SPP communication, hierarchical state machines, etc.), we analyse the suitability of the language for modelling structural and behavioural adaptations at design- and run-time. We evaluate these features in the context of their current state of support in Papyrus-RT, an Eclipse-based MDE tool for UML-RT recently developed by the Eclipse PolarSys Working Group. The use of UML-RT and Eclipse Papyrus for Real-Time (Papyrus-RT) for different kinds of adaptation is demonstrated using two real-time system case studies.**

## I. Introduction

Self-adaptation mechanisms [1], [2] play a central role when designing Distributed Real-time Systems (DRTSs), specifically in critical areas such as automotive or telecommunication, where systems are subject to rapid changes in their surrounding environment or rapid evolution of their requirements. This requires support for automatic reconfiguration of systems at run-time in order to adapt them to changes in their context. To support adaptability at run-time, a Self-Adaptive Software (SAS) system requires a set of properties called self-* properties [1], [3], [4], used to specify the degree of adaptability that the system requires. Examples of these properties are "*self-configuring*" which deals with automatically reconfiguring the system in response to changes, and "*self-optimizing/self-adjusting*" which deals with automatically tuning system performance in response to changes in load.

In SAS systems, highly dynamic environments are complicated to specify, develop, validate, and to verify. Moreover, when the number of configurations an adaptive system can adopt is large, the adaptive system often becomes too complex to be specified and verified directly. In this case, Model-Driven Engineering (MDE) techniques can be applied to manage the increased design complexity posed by the adaptation process and increase reliability. MDE is a model-centric framework that uses models as the primary artifacts in the software development process [5], [6]. Models of a system can be used at both design-time and run-time to provide an abstract, precise and unambiguous representation of the system. Models can help to model the adaptation behaviour of systems independently of the functional behaviour. This can reduce intertwining of adaptive and non-adaptive behaviour, which can significantly decrease the complexity of specifying the adaptation scenarios. In addition, the models can be analyzed before the actual functionality is implemented. This allows discovery of design flaws and inconsistencies in system specifications early in the software development cycle. Thus errors can be corrected more easily and at lower cost, and important issue for safety-critical applications.

Despite a strong interest in modelling SAS systems for real-time, few evaluations of modelling languages exist to assess their suitability for providing an appropriate set of concepts for modelling adaptations. Consequently, it falls to the end-user to evaluate and compare different modelling solutions.

In this work, we evaluate the suitability of UML for Real-Time (UML-RT) for modelling SAS systems. We explore the basic and advanced features of UML-RT suitable for modelling structural and behavioural adaptations. Each feature is evaluated with respect to the different adaptations it can cover at design- and run-time. In addition, we relate these features to their current state of support in Papyrus-RT, an Eclipse/Papyrus based development environment for UML-RT.

**Paper structure:** The rest of this paper is structured as follows: Section II presents the state of the art in self-adaptation and introduces UML-RT. Sections III to V evaluate the suitability of UML-RT and Papyrus-RT for modelling adaptations and discusses current limitations. Section VI details related work, and Section VII concludes.

## II. Background

### A. Self-adaptive Systems

In many real-time systems, a software or hardware component failure could cause injury, financial loss, system thrashing problems, or environmental impact. Self-adaptivity can help these systems react to changing environmental conditions or recover from system failures during execution. Self-adaptation is often designed and implemented using a closed-loop with feedback from the system and environment (i.e., the part of the external world with which the system interacts [2]) [7]. In autonomic computing, this control loop is called MAPE-K, and includes *monitoring*, *analyzing*, *planning*, and *executing* functions [3]. All phases of the control loop access a shared knowledge model of the managed system, its goals and environment. *Monitoring* is responsible for collecting relevant information about the status of the system and its surrounding environment. *Analyzing* is responsible for evaluating the collected data and checks for violations of requirements. In the case of a violation, the *planning* function triggers plans for

strategy adaptation that are executed by the *executing* function. The following focuses on the different kinds of adaptations that are possible to execute during the *executing* function.

TABLE I
TAXONOMY OF ADAPTATIONS

| | | | |
|---|---|---|---|
| Structural | Static | Component | Create / remove / update |
| | | Connector | Create / remove / update |
| | Dynamic | Component | Create / remove / update |
| | | Connector | Create / remove / update |
| Behavioural | Static | Component | Behaviour adaptation |
| | | Connector | Behaviour adaptation |
| | Dynamic | Component | Behaviour reconfiguration |
| | | Connector | Message re-routing |

Table I shows a taxonomy of adaptations that are relevant to modelling real-time systems. Adaptations can be structural, behavioural, or a combination of the two [8]. Structural adaptation aims to adapt system behaviour by changing the system's architecture or environmental constraints. Structural SAS systems support fundamental operations such as adding, deleting, and updating new or existing components and their interconnections [9].

Behavioural adaptation focuses on changes to the functionality or interaction of the computational entities. Because establishing a taxonomy of all possible behavioural adaptations may be difficult and the range of formalisms to express behaviour (e.g., code, discrete models, equation-based systems) is wide, we will narrow our study to modelling behaviour using transition systems for the purposes of this paper. Transition systems can be used to represent system behaviour in terms of states and transitions. One of the most widely used state transition formalisms is Harel's Statecharts [10].

Different techniques exist for adapting the system behaviour. Run-time reconfiguration is a fundamental concept inspired by programming techniques such as aspect-oriented programming, "monkey patching" techniques in JavaScript / Python, and bytecode manipulation. It consists of modifying the behaviour of the system in response to requirement or environmental changes that were not expected when the system was initially designed. One relevant work introducing run-time reconfiguration capabilities is the use of traits in MDE [11].

To further refine the classification in Table I, we differentiate between *static* and *dynamic* adaptation: static adaptation allows reconfiguation of the system during deployment or code generation, whereas dynamic adaptation supports reconfiguration at run-time.

### B. UML for Real-Time

UML-RT [12], [13] is a profile for UML specifically designed for modelling real-time systems with soft real-time constraints. It has its roots in the definition of Real-time Object-Oriented Modeling (ROOM) [14], a domain-specific language for the development of real-time systems, initially adopted by ObjecTime [15]. UML-RT has a long and successful track record of application and tool support in, for example, IBM Rational RoseRT, IBM RSA-RTE [16], and more recently Papyrus-RT [17].

UML-RT features a rather small set of concepts. The main concept is the *capsule*, an active class which owns a *state machine* and can exchange *messages* through its *ports*. Ports are typed with *protocols*, a formal definition of the incoming and outgoing messages a capsule can send or receive. Ports can be connected through *connectors* if their ports are typed with the same protocol. In addition, UML-RT provides only two diagrams to represent the structural part and the behavioural part of a system: *capsule diagrams* representing how capsules are instantiated and inter-connected, and *state machine diagrams* for modelling the behaviour of each capsule.

While the main concepts of UML-RT are rather simple, the language also features more advanced constructs which make it a good candidate for modelling SAS systems. Examples of such constructs include Service Access Point (SAP) and Service Provision Point (SPP) ports to dynamically bind service providers to clients, capsule and state machine inheritance, optional capsules which are dynamically created and wired at run-time, and so on. These features allow UML-RT to support some structural and behavioural adaptations at both design- and run-time.

## III. EVALUATION OF UML-RT

This section presents a systematic review of the UML-RT features suitable for modelling SAS systems. We illustrate these features using a running example, a *Failover system*.

### A. Running Example: a Failover System

To assess the suitability of UML-RT for supporting the different adaptations summarised in Table I, we introduce the *adaptive failover system* as a case study. This system involves a set of server components to handle client requests. It relies on either passive or active replications [18], two common strategies for maximizing availability when building real-time distributed fault-tolerant systems. In passive replication, one server component works as a *master*, handling all the client requests while the *backup* servers are largely idle. Passive replication does not create run-time overhead, except for handshake operations, and for receiving state updates from the master [19]. In active replication, the clients requests are multicast and can be served by all service components. In case of server failure, the remaining servers can continue to provide the service to the clients, which leads to faster failure recovery.

We use a load-balancing scenario to evaluate the support of UML-RT for dynamic adaptations. In this scenario, each server component can only handle a limited number of requests. In that case, the system has to constantly adapt its available resources by dynamically adding or removing server components in response to workload changes.

The following discusses the support and the pertinence of different UML-RT structures for modelling structural and behavioural adaptations at both design- and run-time.

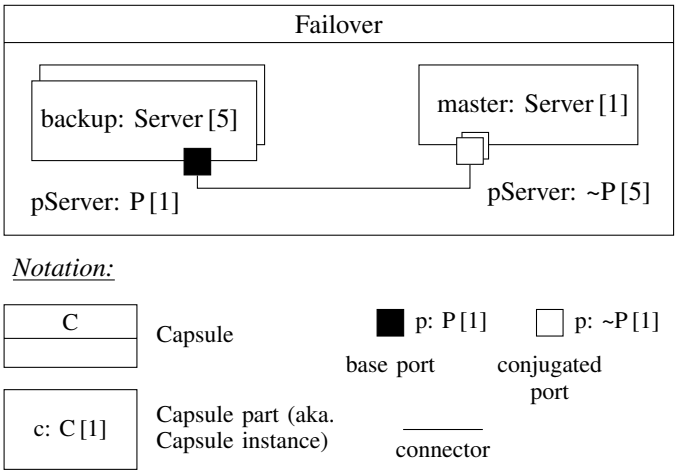


Fig. 1. Failover Model with Five Backup Servers (Static Replication)

### B. Structural Adaptations

Static adaptations are usually well addressed by component-based modelling languages, including UML-RT. Replication is supported by the concept of *multiplicity*, coming from UML. In UML-RT, a capsule is responsible for the life cycle of all capsule parts it composes, from their creation to their destruction. A multiplicity can be defined for a capsule part in order to instantiate it multiple times. To illustrate this concept, let us consider the simple UML-RT model in Fig. 1. It shows a master server connected to five backup servers in order to balance the client request workload between multiple servers. Replication is statically defined in the model through the multiplicity set for the backup capsule part. The port of the master server has to be replicated as well, and should match the multiplicity of the connected capsule part. In UML-RT, replication of either capsule parts or ports is denoted using graphical duplicates.

UML-RT can also support dynamic replication. The central concept is the *optional* capsule. An optional capsule is a capsule whose incarnation is not statically set in the model. It can be useful in the load-balancing scenario, where the number of servers to replicate is not known at modelling time and evolves at run-time according to the workload. Another important concept is the *plug-in* capsule. It acts as a *placeholder* which can be dynamically imported and shared by different capsules. However, due to the limited space, this concept will not be further investigated.

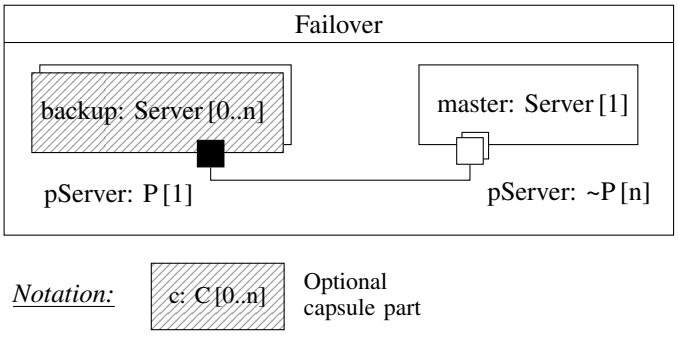


Fig. 2. Failover Model with Several Backup Servers (Dynamic Replication)

Fig. 2 illustrates how optional capsules are modeled in UML-RT. It shows one master server connected to up to $n$ backup servers (with $n \in \mathbb{N}_{>0}$).

The concept of optional capsules has two limitations. The maximum number of backup servers $n$ must be statically set at design-time. In addition, there is no concept of *optional connectors* in UML-RT. Instead, UML-RT has a feature for modelling non-statically wired connectors, normally used for modelling interfaces between applications and platforms [20]. It is based on two special ports, respectively named SAP and SPP, which allow for automatically binding capsules at run-time. In the context of the Failover system, this feature can be used to support the dynamic load-balancing scenario.

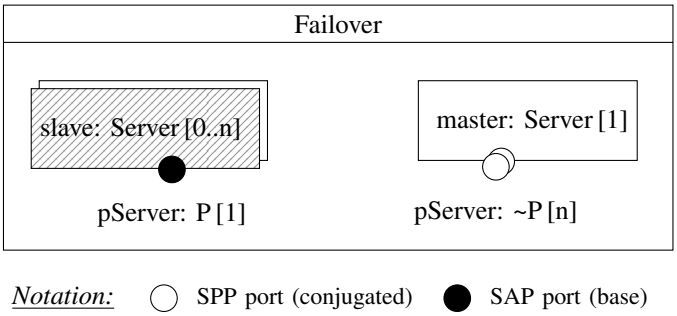


Fig. 3. UML-RT SAP / SPP communication

Fig. 3 shows how the previous model can be adapted by using the SAP and SPP ports. SAP and SPP ports are denoted using the boundary circle notation. No connector is explicitly defined between them, as the connection is created at run-time. To make a SAP / SPP connection work, two assertions must hold: both ports must be typed with the same protocol, and both must have the same name.

### C. Behavioural Adaptations

SAS systems can also involve behavioural adaptations. UML-RT relies on *hierarchical state machines* to model a capsule's behaviour. UML-RT state machines use a simplified version of UML state machines [15]. In UML-RT, states can be composite, containing substates. The hierarchical structure of state machines is an important feature of UML-RT which improves the modular development of SAS system behaviours. In the context of the Failover system, hierarchical states can be used to structure the behaviour of the server components, whether they are master or backup servers (cf. Fig. 4). For example, whenever a failure in the master server occurs, the system should be able to designate another server to become the new master.

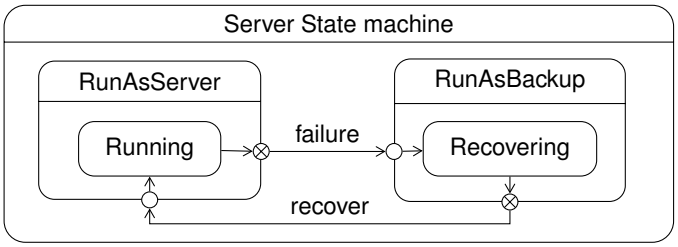


Fig. 4. UML-RT Hierarchical State Machine

Another way of implementing behavioural adaptations at design-time in UML-RT relies on the inheritance mechanism. Widely used in code-based programming for improving modularity, inheritance is also a key concept in modelling languages. UML-RT supports inheritance for capsules, protocols, and state machines. Capsule and protocol inheritance improves modularity and increase reuse of the structure of the model. State machine inheritance is used for extending a state machine or parts of a state machine (such as composite states). To model the Failover system, inheritance can be used, for example, for modelling master and backup server capsules as extensions of an abstract server capsule.

While UML-RT provides reasonable support for static adaptation, it lacks any features for dynamic adaptation. Re-routing of messages is essential whenever a structural adaptation is triggered, or a component enters a specific state where some messages are not expected to be received. Let us consider the following scenario: after a failure is detected, a master server enters a Failure state where no message from the clients is expected. Variability introduced by the concurrent nature of the system may imply that a message is sent by the client at the precise time the master server enters the state. While UML-RT can model this adaptation using concurrency control mechanisms such as *defer/recall*, it does not provide any easy and direct way to model the adaptation.

In response to changes in the system requirements or to face an unexpected situation, reconfiguring the behaviour at run-time is an interesting adaptation to address. For example, one could want to reconfigure the behaviour of the *RunAsServer* composite state dynamically. Programming languages with dynamic typing, such as JavaScript or Python, support this kind of adaptation. To our knowledge, the only application in MDE environments is the work described in [11]. UML-RT does not provide any high-level feature to ease this adaptation.

## IV. Tool Support

Modelling tools influence the ability to represent SAS systems with respect to the ease with which relevant information about the run-time environment and state can be collected and observed. In this section, we discuss the support for various adaptations provided by Papyrus-RT [15], [17], an open-source MDE environment for UML-RT, recently developed by the PolarSys Eclipse Working Group [21]. It allows the generation of complete, executable code from models and advances the state-of-art with support for model representation with mixed graphical/textual notations and an extensible code generator. Papyrus-RT is based on the Papyrus platform [22], [23] and was designed to be extensible, allowing users to add, with relative ease, their own customizations or extensions. Its target audience is industrial developers who want to build custom solutions, researchers who want to prototype and evaluate new techniques, and educators who want to teach students the strengths and weaknesses of modelling and MDE.

We implemented two different use cases in Papyrus-RT, respectively covering the scenarios of static replication and dynamic replication with support for load-balancing[1].

### A. Static Replication (w/o Load-balancing)

The first use case is a simulator illustrating how clients and servers communicate via a Failover system that handles possible failures of the servers and triggers static adaptations. The top-level capsule is shown in Fig. 5. It consists of a simulator capsule containing two servers and five clients. The ENV capsule simulates the environment and keeps track of the configuration information, such as the replication mode (active or passive replication) and the list of master servers. In addition, this capsule is responsible for monitoring possible failures of the servers and providing to any other capsule the current running state of the system.

Static replication is represented by the number of clients connected to the two servers. The multiplicity of the ports of the different capsules have been set to match the number of capsule parts connected to them. Communication with the ENV capsule is implemented using SAP/SPP to dynamically bind the different ports.

Fig. 6 illustrates how inheritance can be used to define both the client and the server capsule by inheriting from the *Host* capsule. It shows a classical UML class diagram, as inheritance is not graphically supported by capsule diagrams. Unfortunately it is not possible to model the state machines of the Server and Client using inheritance from the Host,

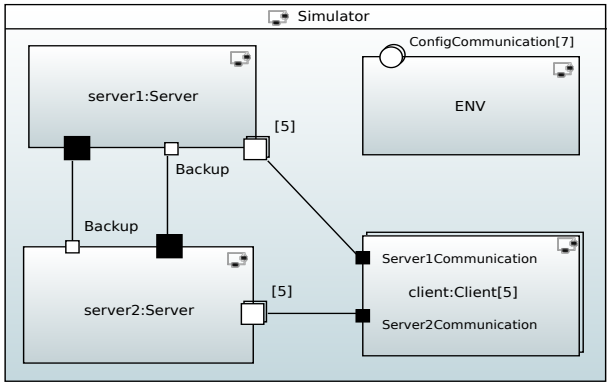


Fig. 5. Simulator Capsule

[1]The two use cases are available here: https://bitbucket.org/kahani/umlrt-self-adaptation-usecases

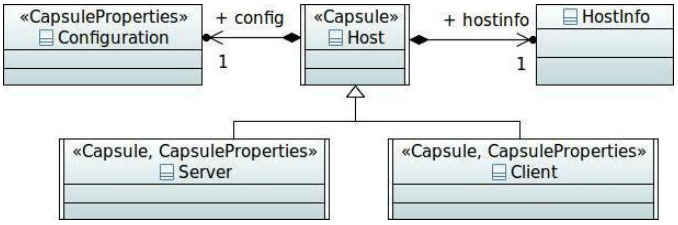


Fig. 6. Host Class Diagram

since Papyrus-RT does not yet support behavioural inheritance (although that support is planned for future versions).

Fig. 7 shows the behaviour of the server capsule. It consists of two main states, where the server can run as either a master or a backup server. A failure state simulates the failure of the master server and is reached after a certain time is randomly computed. When the server recovers from a failure, it may restart as either a master or a backup server, depending on the replication mode and other parameters. The tasks of a master server and a backup server are different. The master server is required to update its state by sending two kinds of messages: *IAmAlive* (sent to the backup servers), and *IAmMaster* (sent to the environment capsule). If it fails in sending these messages, its execution is considered to have failed and a new server must be ranked up. It is also responsible for receiving and processing client requests. The backup server is largely idle, waiting to be ranked up whenever the master server fails. All timeouts are randomly calculated to introduce variability in the execution of the system.

To model the behaviour, we used advanced features of UML-RT that are supported by Papyrus-RT. For example, composite states can be used to ease the modelling of the different behaviours a server may have depending on whether it is a master or a backup server. For modularity purposes, a recent feature of Papyrus-RT creates a state machine diagram per composite state in order to describe its internal structure[2].

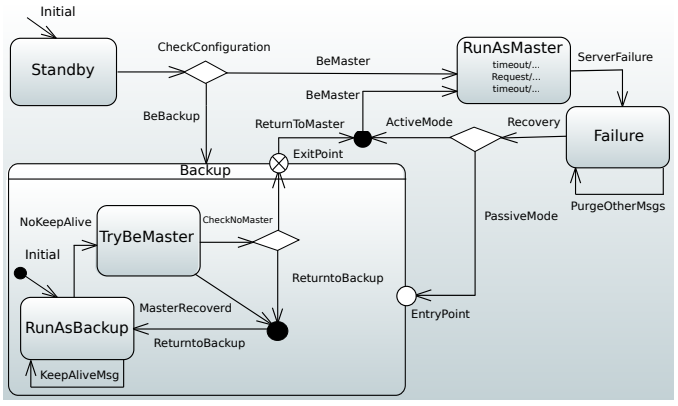


Fig. 7. Server State Machine Diagram

[2]For the sake of simplicity, Fig. 7 shows sub-vertices of the Backup composite state directly embedded in the state machine diagram.

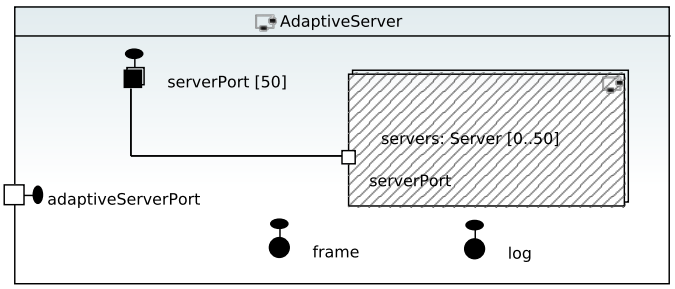


Fig. 8. Adapative Server Capsule Diagram

Papyrus-RT supports the *defer/recall* concurrency control mechanism, and Fig. 7 also shows another workaround that has been used to model message deferring. The Failure state owns a self-transition to catch all the messages coming from a specified port in order to execute an exception routine.

### B. Dynamic Replication (with Load-Balancing)

The second use case demonstrates the suitability of Papyrus-RT for modelling run-time adaptations. It simulates a Failover system allocating servers dynamically in order to load-balance a changing workload. Fig. 8 shows its structure. It consists of an *AdaptiveServer* capsule responsible for adding or removing servers at run-time in response to workload change. A second capsule[3] simulates the generation of a workload that may change over time. The internal structure of the adaptive server consists a list of servers whose the maximum number of allowed replications has been set to 50. The server capsule part is optional, meaning that incarnation and destruction of its instances are delegated to its containing capsule at run-time.

Papyrus-RT provides a fine-grained control of the incarnation and the destruction of optional capsules in the model through the use of the *Frame* protocol. It is a specific service provided by the RTS library, which manages the execution of the system. It provides two system methods: `incarnate` in order to manipulate the creation and binding of an optional capsule and `destroy` for its destruction. To be used, a *frame* service port has to be instantiated into the containing capsule. In addition, the Papyrus-RT RTS library implements an abstract protocol called *UMLRTBaseCommProtocol*, from which all user-defined protocols implicitly inherit. It consists of two messages, respectively `rtBound` and `rtUnbound`, sent whenever a connector between two capsule parts is created or destroyed. These messages can be used to instantiate and destroy servers.

## V. Discussion

Table II summarises the suitability of UML-RT and Papyrus-RT for modeling SAS systems. Both the language and the tool provide reasonable support for modeling behavioural and structural adaptations of SAS systems at design-time and run-time. However, there are some limitations that still need to be addressed. The following summarises them.

[3]Not represented in Fig. 8.

TABLE II
SUMMARY OF ADAPTATIONS SUPPORTED BY UML-RT AND PAPYRUS-RT

| Adaptation | | | UML-RT Structure | Papyrus-RT Support |
|---|---|---|---|---|
| Structural | Static | Component<br>Connector | Replication | Supported by the Run-Time Service (RTS) library |
| | Dynamic | Component<br>Connector | Optional capsule<br>SAP / SPP Communication | Frame service port<br>SAP / SPP service ports |
| Behavioural | Static | Component<br>Connector | Hierarchical state machines ; State machine inheritance<br>Protocol inheritance | Graphical notation ; Partial support of inheritance<br>Partial support |
| | Dynamic | Component<br>Connector | No support<br>No support ; some workarounds exist | No support<br>Defer/recall mechanisms |

Although the process of adding and removing new/existing capsules and their connectors is pretty straightforward, it suffers from some drawbacks. Connectors alone cannot be updated in order to re-configure the system at run-time. This scenario is encountered when the master server fails, requiring another server to replace it. Messages from the client have therefore to be re-routed to the new master server. In our first use case, we statically connected the client capsule to each server and the environment capsule provided the current configuration to the client. A feature could support to dynamically connect components satisfying certain properties. In addition, the *Frame* service is limited to the incarnation and destruction of optional capsule parts and could not be used for e.g., adding parts or attributes that were not conceived at design-time.

A related issue is the lack of message routing support. The RTS library does not provide a service to block ports. Therefore, other capsules can send messages to ports that remain opened. For example, in our first use case, ports of the slave server should be closed for preventing clients from sending messages to them. In this case, the server is not able to process and respond to the requests, which leads to discarding the messages. This scenario is not acceptable in many of real-time systems such as communication systems. A workaround has been shown, but their is no automatic support to do it.

While the present work focuses on adaptations that can be done during the *Executing* phase, it is worth mentionning that Papyrus-RT lacks support for implementing the other phases. For example, the possibility of extracting valuable information of the capsule running state during the *Monitoring* phase (e.g., the message queue load factor) is missing. It required us to manually implement an adaptivity layer responsible for triggering the different adaptations. This layer was designed based on run-time information collected from the different capsules to make adaptation decisions possible.

## VI. RELATED WORK

The evaluation of several aspects of modelling languages, such as MechatronicUML [24], for modelling of SAS systems has been studied over the years. We are the first, to the best of our knowledge, to study the effectiveness of different UML-RT features in modelling of SAS systems. In this section, we discuss the most relevant related work.

The MechatronicUML modelling language, which adopts concepts of the UML, has been used in several publications [25], [26]. MechatronicUML supports the development of structural and behavioural aspects of mechatronic software. It adapts a component-based approach to software structure and changes, and uses real-time statecharts, which are a combination of UML state machines and timed automata, for the specification of the components' behaviour [24]. MechatronicUML uses graph transformation rules represented by story patterns to formally describe behavioural adaptations. It provides features, such as continuous ports and dynamic port addition/removal not supported by UML-RT.

Matlab/Simulink and Stateflow are other modelling languages that have been used to address the modelling of SAS systems. A Simulink model is a hierarchical representation of the system design using a set of blocks interconnected by lines. Simulink blocks allow specification of continuous behaviour [27], which is not supported by Papyrus-RT. However, message-pools or buffers are not supported in Stateflow [24]. Trapp [28] has used Matlab/Simulink to model behavioral adaptation of automotive systems. Based on architecture, adaptation, and functional models, they generate an integrated Simulink model including the functionality and the adaptation behaviour. In another work, Bartosinski [29] uses Matlab/Simulink for modelling, visualization, and debugging of a self-adaptive computing networked entity (SANE). Mars (Methodologies and Architectures for Runtime Adaptive Systems) is another domain-specific modelling language [28] that can be employed to model adaptation behaviour (e.g., used by the approaches such as [28], [30]). For example, Schaefer [30] proposes an integration framework for behavioral adaptation as a semantics-based backend for MARS for model-based verification of adaptive embedded systems. Other approaches, such as HyROOM/HyCharts [31], Masaccio [32], and Charon [33], can also model complex systems using statecharts. A survey of these approaches used for developing software-intensive systems can be found in [27].

Other implementations of UML-RT exist: IBM Rational Software Architect RealTime Edition (RSA-RTE) [16] and IBM Rational RoseRT [34] can both be used to model SAS systems. Both tools support different features, such as behavioural inheritance. Other modelling tools, such as

AutoFOCUS [35] and Modelica [36] can be used to specify and model SAS systems. For example, Autofocus mechanisms, such as system structure diagrams can be used to model system structure, and state transition diagrams to model the behaviour of system components.

## VII. Conclusion

This paper is an evaluation of UML-RT for modelling SAS systems. It proposes a classification of the different structural and behavioural adaptations that are relevant at design- and run-time for modelling real-time systems. For each adaptation, different UML-RT concepts and their support in Papyrus-RT were reviewed in order to assess their suitability for modelling the adaptation. The evaluation showed that UML-RT provides a reasonable set of features for addressing most of the adaptations, while some limitations of UML-RT and its supporting tools still exist, specially for modelling of behavioural adaptations.

The two use cases we implemented also showed that the Papyrus-RT RTS library lacks support for implementing the different phases of the control loop usually implemented for monitoring and triggering the different adaptations of SAS systems. Future work we are investigating includes formalising and implementing such control loops in Papyrus-RT.

## Acknowledgments

This work is supported by Ericsson Canada, EfficiOS, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

## References

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[2] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, D. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "software engineering for self-adaptive systems: A research roadmap," *(Eds.): Self-Adaptive Systemsed*, pp. 1–26, 2009.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] Ö. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, P. Moorsel, and V. M. Steen, "Self-star properties in complex information systems: Conceptual and practical foundations," in *Lecture Notes in Computer Science*, 2005.

[5] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[6] N. Kahani and J. R. Cordy, "Comparison and evaluation of model transformation tools," in *Technical Report 2015-627*, 2015, pp. 1–42.

[7] IBM, "An architectural blueprint for autonomic computing," in *White paper*, 2006.

[8] C. Hofmeister, "Dynamic reconfiguration," Ph.D. dissertation, Univ. of Maryland, College Park, 1993.

[9] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, 2004, pp. 28–33.

[10] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[11] V. Abdelzad and T. C. Lethbridge, "Promoting traits into model-driven development," *Software & Systems Modeling*, pp. 1–21, 2015.

[12] B. Selic, "Using UML for modeling complex real-time systems," in *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, 1998, pp. 250–260.

[13] E. Posse and J. Dingel, "An Executable Formal Semantics for UML-RT," *Software & Systems Modeling*, vol. 15, no. 1, pp. 179–217, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10270-014-0399-z

[14] B. Selic, G. Gullekson, and P. T. Ward, *Real-time object-oriented modeling*. John Wiley & Sons New York, 1994, vol. 2.

[15] E. Posse, "PapyrusRT: modelling and code generation," in *Workshop on Open Source for Model Driven Engineering (OSS4MDE'15)*, 2015.

[16] IBM, "IBM Rational Software Architect RealTime Edition, v9.5.0 Product Documentation," 2015.

[17] "Papyrus for real time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, accessed: 2016-03-10.

[18] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *IEEE*, vol. 30, no. 4, pp. 68–74, 1997.

[19] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive failover for real-time middleware with passive replication," in *Real-Time and Embedded Technology and Applications Symposium*, 2009, pp. 118–127.

[20] B. Selic, "Accounting for platform effects in the design of real-time software using model-based methods," *IBM Systems Journal*, vol. 47, no. 2, pp. 309–320, 2008.

[21] "PolarSys Working Group Homepage," https://www.polarsys.org/, accessed: 2017-01-20.

[22] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier, "Papyrus UML: an open source toolset for MDA," in *Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*, 2009, pp. 1–4.

[23] F. Bordeleau and E. Fiallos, "Model-based engineering: A new era based on papyrus and open source tooling," in *OSS4MDE@ MoDELS*, 2014, pp. 2–8.

[24] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, S. Thiele, W. Schäfer, M. Meyer, U. Pohlmann, C. Priesterjahn, and M. Tichy, "The MechatronicUML design method-process and language for platform-independent modeling," in *Technical Repeport tr-ri-14-337*, 2014.

[25] C. Heinzemann, J. Rieke, and W. Schäfer, "Simulating self-adaptive component-based systems using matlab/simulink," in *SASO*, 2013, pp. 71–80.

[26] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schfer, M. Meyer, and U. Pohlmann, "The MechatronicUML method: model-driven software engineering of self-adaptive mechatronic systems," in *In Companion Proceedings of the 36th International Conference on Software Engineering*, May 2014, pp. 614–615.

[27] H. Giese and S. Henkler, "A survey of approaches for the visual model-driven development of next generation software-intensive systems," *Journal of Visual Languages and Computing*, vol. 17, no. 6, pp. 528–550, 2006.

[28] M. Trapp, R. Adler, M. Förster, and J. Junger, "Runtime adaptation in safety-critical automotive systems," in *Software Engineering*, 2007, pp. 1–8.

[29] R. Bartosinski, M. Danek, P. Honzk, and J. Kadlec, "Modelling self-adaptive networked entities in matlab/simulink," in *Technical Computing Prague*, 2007, pp. 1–8.

[30] I. Schaefer and A. Poetzsch-Heffter, "Compositional reasoning in model-based verification of adaptive embedded systems," in *IEEE International Conference on Software Engineering and Formal Methods*, 2008, pp. 95–104.

[31] T. Stauner, A. Pretschner, and I. Péter, "Approaching a discrete-continuous uml: tool support and formalization," in *Proceedings of the UML2001 Workshop on Practical UML-Based Rigorous Development MethodsCountering or Integrating the eXtremists*, 2001, pp. 242–257.

[32] T. Henzinger, "Masaccio: a formal model for embedded components," in *Proceedings of the First IFIP International Conference on Theoretical Computer Science (TCS)*, 2000, pp. 549–563.

[33] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical hybrid modeling of embedded systems," in *First Workshop on Embedded Software*, 2001.

[34] "IBM Rational RoseRT," http://www-01.ibm.com/support/docview.wss?uid=swg24016586, accessed: 2017-01-20.

[35] "AutoFOCUS," http://af3.fortiss.org, accessed: 2017-01-20.

[36] "Modelica," https://www.modelica.org, accessed: 2017-01-20.