An AsmL Semantics for Dynamic Structures and Run Time Schedulability in UML-RT

Stefan Leue[1], Alin Ştefănescu[2⋆], and Wei Wei[1]

[1] Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
{Stefan.Leue|Wei.Wei}@uni-konstanz.de
[2] SAP Research CEC Darmstadt
Bleichstr. 8, D-64283 Darmstadt, Germany
alin.stefanescu@sap.com

Abstract. Many real-time systems use runtime structural reconfiguration mechanisms based on dynamic creation and destruction of components. To support such features, UML-RT provides a set of modeling concepts including optional actor references and multiple containment. However, these concepts are not covered in any of the current formal semantics of UML-RT, thus impeding the testing and formal analysis of realistic models. We use AsmL to present an executable semantics covering dynamic structures and other important features like run time schedulability. The semantics is parametrized to capture UML-RT semantics variation points whose decision choices depend on the special implementation in a vendor CASE tool. We have built several various implementations of those variation points, including the one as implemented in the IBM Rational Rose RealTime (Rose-RT) tool. Finally, we illustrate how the proposed executable semantics can be used in the analysis of a Rose-RT model using the Spec Explorer tool.

Key words: UML-RT, dynamic structures, formal semantics, AsmL, Rose-RT, Spec Explorer, model-based testing, model checking

1 Introduction

UML-RT [28] was proposed as a UML dialect customized for the design of distributed embedded real-time systems [26]. UML-RT is based on the ROOM notation [27] which was originally developed at Bell Northern Research. Currently supported by the IBM Rational Rose RealTime (Rose-RT) tool [23], UML-RT finds applications in a broad range of domains including telecommunications [17], control systems [25, 12], and automotive systems [11]. A UML-RT model consists of a set of concurrent autonomous objects, called actors, that exchange messages with one another through dedicated communication interfaces referred to as ports. A notable feature of UML-RT is the hierarchical and dynamic structure of an actor: An actor may contain a set of sub-actors in its inner structure,

⋆ The work was done while this author was affiliated with the University of Konstanz.

and a sub-actor can be dynamically constructed and destroyed at run time. Moreover, a sub-actor contained in one actor can be imported to the inner structure of another actor. This allows two actors to share a sub-actor serving as a messenger for its two containers. The dynamic structure feature of UML-RT is very useful since it reflects the architecture of many realistic distributed systems.

Software models play a central role throughout the whole life cycle of development processes following the model driven architecture paradigm [21]. Models are used for documentation, prototyping, code generation and testing. It is therefore of great importance that a software model is correctly designed. A promising way of increasing one's confidence in the correctness of a software model is the use of formal methods, in particular systematic state space exploration. This requires a formal operational semantics of the modeling language.

In this paper we report on the SURTA (**S**emantics of **U**ML-**RT** in **A**smL) project that proposes an executable semantics for UML-RT. The semantics is given in AsmL, a modeling language based on the theory of Abstract State Machines [2]. Developed by the Foundations of Software Engineering group at Microsoft Research, the AsmL language is supported by the Spec Explorer tool [29]. SpecExplorer enables the simulation, assertion checking, and test case generation of an AsmL model by exploring the generated finite state machine. AsmL is also tightly integrated into the Microsoft .NET framework. We use the .NET type system for describing meta-model level details of UML-RT. A further benefit of choosing AsmL is to exploit the verification and test case generation capabilities of Spec Explorer for Rose-RT models, for which the Rose-RT tool offers no support.

Compared to other existing semantics work for UML-RT, the main contributions of the SURTA project are as follows:

- We cover some important UML-RT features such as dynamic structures and run time schedulability. These include (1) actor incarnation/destruction, (2) actor importation/deportation, (3) dynamic port binding, (4) replications, (5) controllers, and (6) other run time environment features.
- We implemented several variants of the UML-RT semantics. To accommodate the many ambiguities and semantic variation points in the semantics as described in [27] we define a most general semantics that can encompass all interpretations that are possible according to [27]. For the non-ambiguous portion of UML-RT we give a parameterized semantics which can extend the general semantics. We also define a semantics that is in line with the Communicating Finite State Machines (CFSMs) [6] paradigm, and one that corresponds to the concrete semantics of UML-RT as implemented in the Rose-RT tool. Providing a semantics as implemented by a particular vendor tool will prove beneficial if this semantics entails a smaller state space than that allowed by the more general semantics. This allows those portions of the behavior to be disregarded that do not correspond to a behavior implemented by that particular tool and hence by the deployed target system synthesized from the model. This would then result in a more efficient state space exploration.

– The architecture of SURTA has great extensibility for implementing and plugging in different concrete semantics. Additionally, SURTA allows a UML-RT model to be straightforwardly expressed in AsmL. The transformation of a model is no more complex than describing what syntactic elements are included in the model, and this can be fully automated.
– SURTA allows an easy encoding of system properties, which can be checked using the assertion checking feature of Spec Explorer.

Related work. [13] uses a notion of flow graphs into which a UML-RT model is transformed. However, there is no systematic transformation method available. It is also not obvious how such a transformation can improve the functional analysis of UML-RT models. An early approach to model checking RoseRT models is described in [24]. It aims at model checking the C++ code synthesized by RoseRT. This approach is rather inflexible, since it depends on the programming language chosen in the RoseRT models and on the particular code generator used. It also only supports a rather limited set of the syntactic features of RoseRT models. [30] uses labeled transition systems to formalize a subset of UML-RT mostly focusing on the behavior of state machines. [19] covers the timed aspects of UML-RT by translating a timed state machine into a timed automaton. However, it does not consider any other aspects of UML-RT. There are several approaches of formalizing UML-RT using process algebras [10, 9, 22, 8, 3]. Most of these works consider only synchronous communication. [9] considers asynchronous communication, but it allows only the use of *bounded* FIFO message buffers. While none of the above cited papers considers dynamic structures of UML-RT, [3] proposes a semantics for the so-called *unwired* ports using name passing in the π-calculus. The support of unwired ports enables changes in the communication topologies. However, the work described in [3] does not consider other kinds of dynamic structures that our work is addressing. Unlike our work, none of the existing work addresses the ambiguities and semantic variation points in the informal semantics of UML-RT, and most of them derive the operational semantics from the particular implementation in Rose-RT. [18] maps the modeling elements of UML to AsmL data structures, based on which a UML model can be transformed into an AsmL specification at the semantic level. This is different from our approach in which a UML-RT model is translated into AsmL purely syntactically. [5] compares the expressiveness of several formalisms for specifying dynamic system structures. SURTA supports almost all important dynamic structure operations that other formalisms provide, such as addition and removal of elements and connectors, iterative changes, and choice-based changes.

Outline. The paper is structured as follows. We give a brief introduction to UML-RT and AsmL in Sections 2 and 3, respectively. The architecture of SURTA is explained in Section 4. The executable semantics is then detailed in Section 5. We illustrate the usefulness of the given semantics in Section 6. We conclude the paper and suggest future work in Section 7.

2 UML-RT

Using an example model in Figure 1, we briefly introduce the set of UML-RT concepts for which we later define a semantics in Section 5. The model in Figure 1 has a number of clients that each requests a remote service. The request of a client prompts the client manager to send an unused service accessor object to the server. The server connection manager then imports the received object to both the coordination process (`serviceAccessorCoor`) and an available `serviceAccessorS` process. The coordination process informs the client side of a successfully established connection. The `serviceAccessorS` process then uses the imported accessor object to pass messages between the client and the service object assigned to the client.

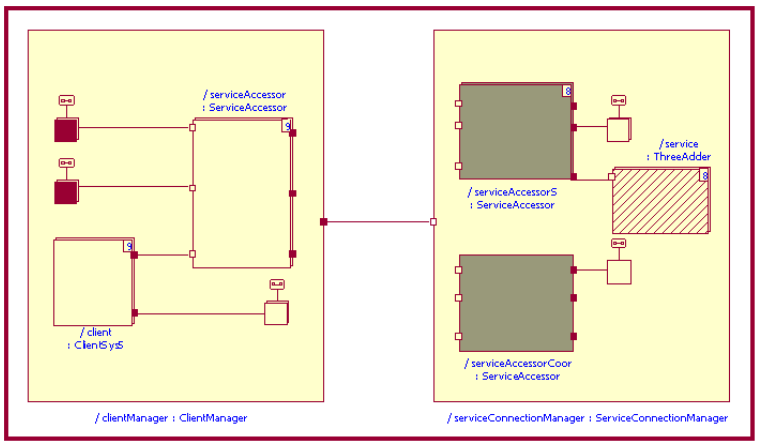

Fig. 1. A UML-RT model in which a number of clients request remote services. The model is taken from a collection of examples included in Rose-RT distributions, with slight modification to allow dynamic creation and destruction of `service` instances.

Static structure. The central structural concept in UML-RT is that of an *actor*, also called capsule in Rose-RT. An actor is an autonomously running object whose execution is influenced by other actors exclusively through message passing. Each actor is an instance of a certain *actor class*. An actor may hold a set of *actor references* (capsule role in Rose-RT), which are pointers to sub-actors. Figure 1 shows the internal structures of two actor classes: `ClientManager` and `ServiceConnectionManager`. Both actor classes contain sub-actor references like `client` and `serviceAccessorS`. A reference is *incarnated* when a new actor is created for the reference. An actor reference is *fixed* if its incarnation occurs when the enclosing actor is created. A reference is *optional* if its incarnation must be explicitly invoked by the enclosing actor. The sub-actor that an optional actor reference points to may be destroyed later by the container. An *imported* reference cannot be instantiated, and can only hold a pointer to an existing

actor. In Figure 1, the reference `client` is fixed and the reference `service`, contained in `serviceConnectionManager`, is optional. Taking the graphic notations of Rose-RT, we denote an optional reference by a shadowed rectangle. Examples of imported references are `serviceAccessorS` and `serviceAccessorCoor`, represented by gray filled boxes.

The communication interface of an actor is a set of *ports* though which the actor sends and receives messages. Each port is associated with a *protocol* that determines which messages can be sent and received through the port. In Figure 1, ports are those small rectangles sitting either on actor reference boundaries or inside actor bodies. A port may be an *end port* through which its containing actor sends or receives messages. A port may also be a *relay port* that simply forwards messages from one side to another. A port may have a *conjugated* protocol, namely, the set of incoming messages and the set of outgoing messages defined in the original protocol are inverse. Ports with conjugated protocols are denoted by hollow rectangles. Ports in Figure 1 are connected with each other. Each connecting line defines a potential binding of the two connected ports. When two ports are actually *bound* at run time, a message can arrive at one port from the other. Two bound ports must have compatible protocols, namely that the set of outgoing messages allowed by one protocol must be contained in the set of incoming messages of the other protocol.

Actor references and ports can be replicated. A replicated entity represents a number of instances. Resembling an array data structure, each individual instance of the entity is accessed through a unique index. Replicated entities are graphically represented in a multilayered fashion, c. f. the actor reference `client` in Figure 1. A replicated entity may have a replication factor to specify the maximal number of instances that it may contain at run time. The replication factor of each replicated reference in our example is depicted as the upper-right corner number in its graphic notation. All ports in the example have also replication factors that are however not shown in the figure. Replication is used to obtain a more flexible and concise graphic representation of a model. However, replication also introduces ambiguities, e.g., when two replicated ports are connected, it is not clear which instance of one port should be bound to which instance of the other port at run time. We will discuss this problem in depth in Section 5.6.

Dynamic behavior. The behavior of an actor is expressed by an *extended hierarchical state machine*. In UML-RT, a transition is always triggered by the receiving of a message from one of the end ports of the corresponding actor. A transition may have a specified action to be executed when it is fired. Actions may alter local variables, send messages to other actors, or dynamically change the structure of the actor. States may also have entry actions and exit actions. The language used for specifying actions is not limited in UML-RT. For more information about state machines and their behavior, we refer readers to [20]. In this paper we consider only flat state machines, and leave the formalization of hierarchical state machines for future work. Currently, we must flatten the hierarchical state machines of a model before translating it into AsmL.

Controllers. At run time, each actor instance runs on a thread. Threads are executed independently in parallel. We abstract physical threads to the concept of *controllers*. A controller contains a set of actor instances, and schedules the executions of contained actors as well as the sending and receiving of messages.

Rose-RT. Widely used in industrial practices and academic research, Rose-RT is a powerful CASE tool for modeling distributed embedded systems using UML-RT formalism [23, 14]. Rose-RT currently supports three programming languages, C, C++, and Java, for specifying transition actions in state machines.

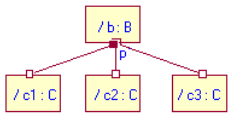

Fig. 2. A UML-RT model.

Rose-RT is often taken as the main source of retrieving an operational semantics for UML-RT. This is problematic in two ways: First, the concrete Rose-RT semantics does not allow for non-determinism at all, and resolves non-determinism in a naive way: Consider the example in Figure 2. The replicated port p is connected to three ports that each belongs to a distinct reference below. Rose-RT decides that the instance of p indexed at 0 will be bound to the port of the first reference being added to the structure diagram when the model was built, say *c1*. This order fixing can be dangerous because a previously validated property may no longer holds for the system after, e.g., the reference *c1* was deleted from the model and later added back. In this case, the instance indexed at 0 will no longer be bound to the port at *c1*, which may result in a violation of the property. Second, the Rose-RT semantics even has some inconsistencies with the informal UML-RT semantics suggested in [27].

3 AsmL

Space limitations do not permit us to provide a detailed description of the Abstract State Machine Language (AsmL in short), we refer the reader to [15] for more information. AsmL is an object-oriented software specification language. Its syntax resembles Java and C#, and it provides conventional object oriented programming features like encapsulation and inheritance. As a high-level specification language, AsmL also provides supports for non-determinism, parallel updates of variables, and assertion checking, as illustrated by the example in Figure 3. The program defines a class of integer intervals, which has two fields to specify the lower bound and the upper bound of an interval. The *shift* procedure shifts an interval by a specified offset. The *random* function returns an arbitrary integer number within the interval.

```
class Interval
  private var lower as Integer
  private var upper as Integer
  public procedure shift(offset as Integer)
    step
      lower := lower + offset
      upper := upper + offset
  public function random() as Integer
    return any number | number in {lower..upper}
  constraint wellformed:
    lower <= upper

Main()
  step
    let interval = new Interval(0, 5)
  step
    interval.shift(6)
    WriteLine(interval.random() > 5)
  step
    WriteLine(interval.random() > 5)
```

Fig. 3. An AsmL program working on integer intervals.

Non-determinism. The returned value of the *random* function is picked non-deterministically from the interval by the run time environment of Spec Explorer, the specification analysis tool supporting AsmL. There are other program constructs in AsmL that allow non-deterministic choices, such as `choose i in S` which takes a random member from a set S.

Parallel updates of variables. The AsmL language has an important concept called `step`. Any update of a variable within one step does not take effect until the next step is executed. As an example, the second step in the *Main* procedure shifts an interval by 6, and checks if any value in the interval is now larger than 5. The checking occurs before the next step is executed, so the interval is still between 0 and 5 and the output of the checking is *false*.

Assertion checking. AsmL allows the use of class invariants, identified by the keyword `constraint`, as well as pre- and post-conditions for procedures and functions, denoted by the keywords `require` and `ensure`, respectively. The violation of any of these assertions will result in a runtime exception. Assertion checking does not only help revealing program errors, but also enables property verification by searching for assertion violations, as explained in Section 6.

Semantics of AsmL. The AsmL language is based on the Abstract State Machine formalism [4]. A formal semantics has been defined for the core of AsmL [15]. There are also other semantic definitions for AsmL [16]. Even if it does not possess a formal semantics for all of its syntactic features, AsmL with its support for non-determinism and its step semantics is nonetheless an ideal choice for defining an executable operational semantics for UML-RT, which is the main objective of our paper.

4 SURTA Architecture

The SURTA project defines an AsmL specification of the UML-RT semantics, including various realizations of semantic variation points. Our objective is to give a semantics for the UML-RT language. We do not consider how the semantics of a concrete UML-RT model is mapped into AsmL structures, i.e., we are not interested in devising a semantic level translation procedure for individual

models. This distinguishes our work from all other semantic work for UML-RT, e.g., [22, 1, 3]. The transformation of a UML-RT model can then be accomplished entirely at the syntactic level: we simply describe what comprises the syntactic definition of a model in the AsmL specification. The execution of the model is then handled totally by the semantics of a chosen run time environment. This approach has at least the following advantages:

– The separation of syntax and run time behavior makes it highly flexible for implementing semantics variants: When a semantic variation point of UML-RT is differently realized, the syntactic level needs no or very little modification.
– Because a concrete UML-RT model is only syntactically transformed into AsmL, a change in the semantic level does not require the model to be re-transformed. In fact, how the semantic level of UML-RT is mapped can be completely transparent to users of the SURTA project.
– A straightforward syntactic translation of a UML-RT model into AsmL can be fully automated.

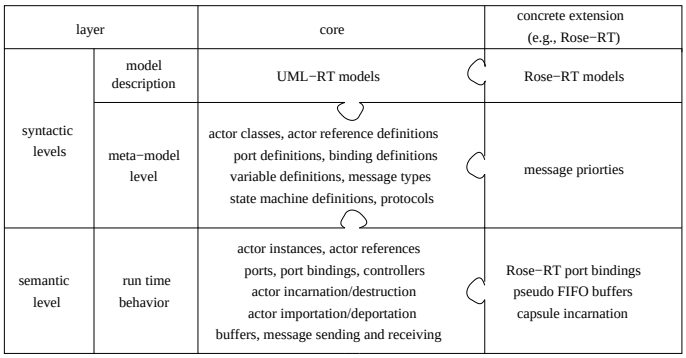

Fig. 4. The SURTA architecture.

Figure 4 shows the architecture of the SURTA project. It has three levels: the syntactic *meta-model level*, the semantic *run-time behavior level*, and the *model description level*. Each level has a core that gives semantics for the unambiguous parts of the UML-RT language, which can be extended by different implementations for ambiguities and semantic variant points. The extensibility minimizes the effort for implementing a variant of some UML-RT semantics.

Meta-model level. This level mainly defines the syntactic composition of each UML-RT modeling concept. For instance, it defines what constitutes an actor class definition. On the other hand, it does not describe what an instance of an actor class is, and how instances of a class behave at run time. The concept of actor instances and its run time behavior are instead defined at the run time semantic level. The meta-model level links the other two levels, and provides indispensable information for actor reference incarnation and importation as will be explained in Section 5.6.

Run-time behavior level. This level defines run time entities that are instances of the modeling elements introduced in the meta-model level. As an example, actor instances are running entities when a UML-RT model is executed, and each running actor instance belongs to some actor class. This level defines the relationship between an actor instance and its defining class, the creation and destruction of an actor instance, port bindings, and many other actor instance related run time properties.

Model description level. This level defines concrete UML-RT models by creating the proper syntactic description of a model based on the relevant UML-RT modeling elements at the meta-model level. The model description level basically needs no knowledge of the run time behavior level. Exceptions are transition guards and actions in which some methods provided by a particular run time environment may be invoked, such as the incarnation of an actor reference, or the read/write of a variable (or, an attribute or a field) of an actor. However, in these cases we only need to know the signatures of the invoked methods while their implementations remain hidden.

Extensions. Each level in the SURTA architecture can be extended with different concrete semantic variation point realizations. Currently, we provide three different implementations: (1) a most general semantics in which each actor instance runs on a distinct thread, and messages are stored in bag-like data structures. This means that any message in a bag can be selected for triggering a transition; (2) a semantics based on Communicating Finite State Machines (CFSMs) [6], in which each actor instance runs on a distinct thread, and messages are stored in first-in-first-out (FIFO) queues such that only the head message can be received by a port; (3) a semantics based on the Rose-RT implementation the details of which will be presented in Section 5.

5 An Executable Semantics

In this section we present the AsmL specification of the UML-RT semantics in the SURTA project. Sections 5.1–5.3 explain how the syntactic definitions of UML-RT modeling elements are mapped. Sections 5.4–5.8 explain in detail the semantics of run time entities. Section 5.9 addresses model descriptions and gives the AsmL specification of the example in Figure 1. Due to space limitations we leave out a large part of implementation code of the definition of each UML-RT concept, and present only the part relevant to our discussion. Readers are referred to [20] for more details.

5.1 Actor Classes

The syntax of the central UML-RT concept of actor classes is defined to be a collection of actor reference definitions, port definitions, binding definitions, variable definitions, and a state machine description (see Section 5.2). This is

reflected in the AsmL class *ActorClass* shown in Figure 5. Note that any object of the class *ActorClass* is a particular class of actors, but *not* an instance of an actor class. Actor instances are defined at the run time behavior level as explained in Section 5.6.

```
class ActorClass
  private name as String
  private const subActorRefDefs as Set of SubActorRefDefinition
  private const portDefs as Set of PortDefinition
  private const bindingDefs as Set of BindingDefinition
  private const variableDefs as Set of VariableDefinition
  private const stateMachine as StateMachine
```

Fig. 5. The AsmL definition of actor classes.

We omit the classes for actor reference definitions, port definitions, etc. The class of sub-actor reference definitions has (1) a field *myClass* to indicate the class of an actor reference; (2) a field *kind* to specify whether a reference is fixed, optional, or imported; and (3) a replication factor which is enforced to be a positive number in our specification. A port definition has one of three types: either an *external end port*, an *internal end port*, or a *relay ports*. An external end port is visible to the outside of the containing actor, and cannot be bound to any port inside the actor. An internal end port is visible only inside the actor, and cannot be connected to the outside world. A port definition has also a replication factor.

5.2 State Machines

As mentioned before, we formalize here only flat state machines. Hierarchical state machines are left for future work. The class of state machines consists of a set of states, transitions, and an initial state. A transition has a source state, a target state, one or more triggers, and an optional action. A trigger consists of a signal and a port. It may also contain a *guard* object so that the trigger is available only if the guard evaluates to true. Note that a state machine is defined only once for a class of actor instances. The class *ActorInstance* does not contain state machine information except for a field to remember its current state. The state machine definition in an actor class guides actor instances to move from states to states.

Guards and actions. We define transition guards and actions as AsmL interfaces. A guard has a mandatory *evaluate* method to be implemented individually for each concrete transition guard. The evaluation of a guard depends on the current state of the respective actor instance and the content of the message used to trigger transitions. Unlike guards, the mandatory method *execute* of an action also takes a run time environment as one parameter. This is because the run time environment provides necessary information for the incarnation, destruction, importation, and deportation of actors, as well as for message sending and receiving.

5.3 Protocol

A protocol is defined to be a set of incoming message types and a set of outgoing message types. A protocol can be conjugated by reversing the two sets of message types. A message type is a pair of a string `signal` and a data type. A message contains a signal, a specified priority, and an optional data object. The signal of a message must be identical to the signal of its message type, and the data object in the message must have the type as specified in the message type.

Rose-RT message priorities. There are seven levels of message priorities defined in Rose-RT: *background*, *low*, *general*, *high*, *panic*, *system*, *synchronous*, from the lowest to the highest. Each priority corresponds to two message buffers in a controller in the Rose-RT run time environment, which we will discuss in Section 5.7.

So far we have been concerned with the meta-model level mappings. Starting with the next subsection we will discuss the run time behavior level that defines a semantics for run time UML-RT entities.

5.4 Actor References

Figure 6 defines the class of actor references. Note that it merely contains a reference to its definition and an instance mapping. As seen in the constructor method of the class, when an actor reference object is created, all fixed sub-actor references are incarnated.

```
class ActorReference
  private const myDef as SubActorRefDefinition
  private var instance as Map of Integer to ActorInstance
  ActorReference(aDef as SubActorRefDefinition)
    myDef = aDef
    match aDef.getKind()
      ActorReferenceKind.fixed:
        instance = {i -> new ActorInstance(aDef.getClass())
                        | i in {0..aDef.getReplicationFactor() - 1}}
      otherwise
        instance = {->} as Map of Integer to ActorInstance
```

Fig. 6. The AsmL definition of actor references.

5.5 Ports

The class *Port* defines actual ports created at run time, as shown in Figure 7, extended by two sub-classes representing two types of ports: end ports and relay ports. A port object records the run time binding information of each of its indexed members. The only difference of the *EndPort* and *RelayPort* classes is that a relay port resides on the structural border of an actor and thus has two sides. One side connects to the outside world of the actor, the other to the inside world. Therefore, a relay port can be involved in more than one binding. We use a Boolean component in the *peer* field to indicate the side of connectivity.

```
class Port
  private const myDef as PortDefinition
class EndPort extends Port
  private var peer as Map of Integer to (Port, Integer) = {->}
class RelayPort extends Port
  private var peer as Map of (Integer, Boolean) to (Port, Integer) = {->}
```

Fig. 7. The AsmL definition of run time ports.

5.6 Actor Instances

Figure 8 shows the fields of the class of actor instances: *myClass* indicates which actor class an actor instance belongs to, the internal structure (sub-actor references and ports), a valuation of the actor variables, and the current state of the actor instance.

```
class ActorInstance
  private const myClass as ActorClass
  private const subActorRefs as Set of ActorReference
  private const ports as Set of Port
  private var valuation as Map of VariableDefinition to Obj = {->}
  private var state as State
  ActorInstance(aClass as ActorClass)
    myClass = aClass
    state = aClass.getStateMachine().getInitialState()
    ports = {new EndPort(pDef) | pDef in aClass.getPortDefs() where pDef.isEndPort()} union
              {new RelayPort(pDef) | pDef in aClass.getPortDefs() where pDef.isRelayPort()}
    subActorRefs = {new ActorReference(rDef) | rDef in aClass.getSubActorRefDefs()}
    step while exists binding in getAllPossibleBindings() where binding.bothPartiesFree()
      choose binding in {b | b in getAllPossibleBindings() where b.bothPartiesFree()}
      binding.bind()
```

Fig. 8. The AsmL definition of actor instances.

Constructor and port binding. Figure 8 also shows how an actor instance is constructed from an actor class definition: A run time port is constructed for each port definition of the class; a sub-actor reference is created for each reference definition of the class. The most intricate part of actor instance construction is how ports inside an actor should be bound, which is only vaguely described in [27]. This is especially the relevant when binding replicated ports or ports of replicated actor references. As a solution, we leave the port binding problem as a semantic variation point. Figure 8 shows the most general binding strategy: when a port p can be bound to either p_1 or p_2, we non-deterministically choose one of p_1 and p_2 to be bound with p.

Rose-RT port binding. Rose-RT fixes the order of port bindings by giving priorities to actor references and ports according to the following rules: (1) For any two sub-actor references r_1 and r_2, if r_1 was added to the model earlier than r_2 was at model construction time, then the ports in the actor that r_1 points to are bound earlier. (2) For a replicated sub-actor reference, for any two indices

$i_1 < i_2$, the ports of the reference member at i_1 are bound earlier than those of the member at i_2. (3) For any two ports p_1 and p_2 in a same actor, if p_1 was added to the model earlier than p_2, then p_1 will be bound earlier. (4) For a replicated port, the member of the port at a smaller index will be bound earlier.

The priority assignment based on indices is natural and easy to control at runtime. However, as discussed in the end of Section 2, priorities based on model element construction time can be dangerous when used during analysis. Therefore, we implement the order of port bindings only with respect to replicated entity indices, and leave other decision choices totally non-deterministic. Such an implementation deviates from the actual Rose RT semantics, and results in a super set of the model behavior that Rose-RT allows. Due to space limitations we omit the details of the implementation of Rose-RT port binding here, which can be found in [20].

Actor reference incarnation and destruction. The method *incarnateAt* of the class *ActorInstance* incarnates a sub-actor reference at a particular index. The method restricts the incarnated reference to be optional. After creating a new actor instance for the reference, the method checks whether there are now ports that need to be bound, and binds them using the most general strategy as described previously. The destruction of an actor reference unplugs all the ports in the actor that the reference at a particular index points to, and then removes the actor pointer from the reference at the index.

Actor importation and deportation. The informal UML-RT semantic regarding actor importation and deportation in [27] results in yet another substantial ambiguity. For an actor to be imported to a reference, it states that the actor must satisfy all the contracts that the reference has with its environment: If there is a binding defined for the reference, then the actor must have a free port that can be bound by this definition. Some confusion is again introduced through the concept of replication. As an example, when a replicated port of an imported reference needs to be bound during importation, it is not clear whether all members of the corresponding port in the imported actor must be free, or only some members of the port need to be free. Our solution, as shown in Figure 9, gives the most general semantics requiring that, for each binding definition that involves the imported actor reference, at least one actual binding can be established by this definition during importation.

Substitutability. The *importAt* method requires the imported actor be of the same class as the imported reference is. This is however unnecessary when substitutability is allowed. In this case, it is sufficient that the imported actor has a compatible set of ports to satisfy the binding contracts of the respective reference. However, the informally described communication interface compatibility gives rise to further ambiguities and confusions. We will address this issue in future work.

```
public procedure importAt(actor as ActorInstance, aRef as ImportedActorReference,
                          index as Integer)
  require actor.getClass() = aRef.getClass() and aRef in subActorRefs
    and not aRef.isInstantiatedAt(index)
  step
    aRef.setInstanceAt(index, actor)
  step
    if (forall bDef in myClass.getBindingDefs holds getAllPossibleBindings(bDef) <> {})
      then step while exists binding in getAllPossibleBindings()
        where binding.bothPartiesFree()
        // bind port here.
    else
      WriteLine("No binding possible. Importation fails.")
      throw new Exception()
```

Fig. 9. The AsmL implementation of actor importation.

5.7 Controllers

We define controllers as an AsmL interface, which has the advantage that various concrete controllers can be implemented. We have implemented three controllers: (1) most general controllers using bag-like data structures for storing messages; (2) CFSM-based controllers using FIFO message queues; and (3) Rose-RT controllers that we discuss in detail in the following. The definitions for (1) and (2) can be found in [20].

Rose-RT controllers. A Rose-RT controller c can host multiple actor instances. It does not however offer a separate FIFO message queue for each end port of each hosted actor. Instead, it builds two queues, shared by all contained actors, for each kind of message priorities. One queue is to store internal messages, i.e., messages whose sender and receiver are both hosted by c. The other queue is for external messages whose sending actor resides in a different controller than c.

Scheduling executions of hosted actors. The *makeStep* method of the class *RoseRTController*, as shown in Figure 10, shows how a Rose RT controller schedules the executions of its hosted actors. The pre-condition of the method requires at least one actor to be executable, by checking whether there is a fireable transition. The principle of searching for fireable transitions is described as follows: It first appends the content of the highest-priority non-empty incoming message queue to the internal queue of the same priority. It then checks the head message of the internal queue of the highest priority. If the message cannot be used to trigger any transition, it checks the queue of the next lower priority. Whenever it finds a message that can be used to trigger (possibly multiple) transitions, the searching terminates and returns the set of fireable transitions triggered by that message. The *makeStep* chooses randomly a fireable transition to execute.

5.8 Run Time Environment

The role of a run time environment is to schedule controllers and to provide model designers with a set of operations such as actor incarnation and message

```
class RoseRTController implements Controller
  public procedure makeStep(re as RunTimeEnvironment)
    require executable()
    choose (actor, transition, buffer) in getFireableTransitions()
      step
        // Receive message here.
      step
        actor.getCurrentState().getExitAction().execute(actor, re)
      step
        transition.getAction().execute(actor, re)
      step
        actor.move(transition)
      step
        actor.getCurrentState().getEntryAction().execute(actor, re)
```

Fig. 10. The AsmL implementation of the Rose RT controller scheduling method.

sending. Figure 11 shows a part of the AsmL interface for run time environments. The last method *run* is used to execute a model. The other methods shown in the figure are functions that can be called in user-defined actions to incarnate/destroy actor references, import/deport actors, send messages, etc.

```
interface RunTimeEnvironment
  sendAt(actor as ActorInstance, portName as String, index as Integer, message as Message)
  incarnateAt(actor as ActorInstance, refName as String, index as Integer,
                        controller as Controller)
  destroyAt(actor as ActorInstance, refName as String, index as Integer)
  importActorAt(actor as ActorInstance, inst as ActorInstance, refName as String,
                          index as Integer)
  deportActorAt(actor as ActorInstance, refName as String, index as Integer)
  run(model as Model)
```

Fig. 11. The AsmL interface of run time environments.

Rose-RT run time environment. Figure 12 shows the implementation of the *run* method in the class *RoseRTRunTimeEnvironment*. Controllers run on separate threads in the Rose-RT run time environment, and the order of controller executions is therefore non-deterministic. Note that every model has one unique capsule class whose only instance at run time is used as the top container for all other capsules in the model. The model execution starts by creating an instance of the top capsule class. Afterwards, the executions of existing controllers interleave, along which new controllers may be created and destroyed. In each interleaving step, the run time environment checks whether there are any executable controllers from which a random one is picked for execution. If no executable controller exists, then the run time environment arbitrarily takes a non-empty message buffer in the system and removes the head message of that buffer. This makes sure that an unused message does not block the availability of other messages in the same queue.

```
class RoseRTRunTimeEnvironment implements RunTimeEnvironment
  private var controllers as Set of RoseRTController
  public procedure run(model as Model)
    step
      let topActorInst = new ActorInstance(model.getTopActorClass())
      let topController = new RoseRTController()
    step
      topController.addActor(topActorInst)
      add topController to controllers
    step while (exists controller in controllers where (controller.executable()
                or controller.containsNonEmptyBuffer()))
      if (exists controller in controllers where controller.executable()) then
        let executables = {controller | controller in controllers where controller.executable()}
        choose controller in executables
          controller.makeStep(me)
      else
        choose controller in controllers where controller.containsNonEmptyBuffer()
          controller.removeSomeHeadMessage()
```

Fig. 12. The AsmL implementation of the Rose-RT run time environment.

5.9 Model Descriptions

The AsmL specification of a concrete UML-RT model is represented by a set of syntactic definitions of model elements based on the meta-model level data structures. A Rose-RT model in AsmL is composed of a set of capsule classes and the top capsule class. The set of protocols is not explicitly presented. As mentioned previously, the transformation of a Rose-RT model into its AsmL definition can be fully automated since it is a direct syntactic mapping. Exceptions are transition actions that can be written in a high level programming language like Java. Due to the different expressiveness and semantics of Java and AsmL, we cannot map the whole Java language into AsmL. However, a large part of Java features and statements can still be straightforwardly and automatically translated to AsmL equivalents. Figure 13 gives partly the AsmL specification of the model in Figure 1.

6 Validating Models with SURTA

With the support of Spec Explorer, we intend to use the UML-RT semantics defined in SURTA to accomplish the following tasks: (1) model checking UML-RT models; (2) simulating UML-RT models for checking potential property violations; and (3) test case generation for model-based testing [7]. Model-based testing is out of the scope of this paper, and we will investigate this in future work. Model checking is impeded by the fact that the currently publicly available version of SpecExplorer is not completely exploring the state space, in particular if the AsmL model includes nondeterminisic choices, as is the case in SURTA code. We hence illustrate how the random walk simulation feature of SpecExplorer can be used to reveal property violations for Rose-RT models.

Consider a simple model of a resource sharing system in which a central server grants exclusive access to a shared object to multiple clients. In the model, a

```
class SCMSRAction implements Action
  public procedure execute(actor as ActorInstance, re as RunTimeEnvironment) as Boolean
    var theClient as ActorInstance = re.getLastReceivedMessage(actor).getData()
                                        as ActorInstance
    step
      re.importActor(actor, theClient, "serviceAccessorCoor")
    step
      re.importActor(actor, theClient, "serviceAccessorS")
    step
      let srMessage = new Message("serviceReady", RoseRTMessagePriority.general)
      re.send(actor, "connectionSetup", srMessage)
    step
      re.deportActor(actor, "serviceAccessorCoor")
// ... ...
// Define the ServiceConnectionManager class
let serviceAPort = new ExternalWiredEndPortDefinition("serviceAccess",
                                                      serviceAccessProt, true)
let service = new FixedCapsuleRoleDefinition("service", threeAdderClass, 8)
let SCMActive = new State("Active")
let sCMSRTGuard = new SCMSRTGuard()
let SCMSRTTrigger = new Trigger("serviceRequest", serviceAPort, sCMSRTGuard)
let sCMSRAction = new SCMSRAction()
let SCMSRTTran = new Transition("serviceRequest1", SCMActive, SCMActive, {SCMSRTTrigger},
                                    sCMSRAction)
let SCMStateMachine = new StateMachine({SCMInit, SCMActive, SCMFull}, SCMInit,
      {SCMInitTran, SCMSRTTran, SCMSRFTran, SCMRSRTran, SCMDeportTran1, SCMDeportTran2})
let activeVar = new VariableDefinition("active", type of Integer)
let serviceConnectionManagerClass = new CapsuleClass(
    "ServiceConnectionManager", {service, serviceAccessorS, serviceAccessorCoor},
    {serviceAPort, connectionTPort, connectionSPort}, {SCMConn1, SCMConn2, SCMConn3},
    {activeVar}, SCMStateMachine)
let System5Class = new CapsuleClass("System5", {clientManager, serviceConnectionManager},
                    {}, {System5Conn}, {})
let capsuleClasses = {System5Class, serviceAccessorClass, clientSys5Class,
                        clientManagerClass, threeAdderClass,
                     serviceConnectionManagerClass}
let model = new RoseRTModel("System5", capsuleClasses, System5Class)
```

Fig. 13. The AsmL specification of the model in Figure 1.

client is possessing the object access when it is in the `operation` state. An important safety property, sometimes referred to as mutual exclusion, is that no more than one client is allowed to have access at any given point in time, i.e., only one client can be at the `operation` state. In SURTA, we encode the negation of the property into the function *multipleAccess* in a *MuTexVerification* class, as shown in Figure 14. This function is checked by the *check* method of the class. The *run* method of the *RoseRTRunTimeEnvironment* is modified to invoke the *check* method after each interleaving step. In this way, the property is checked automatically in every step of the model simulation.

An example for the checking of the violation of a liveness property is the fairness constraint that each client will eventually be granted access to the object. This property is encoded in the *allHadAccess* method in Figure 14. The method always returns true during the simulation, and in the meantime adds a client to the set *hadAccess* when the client reaches the state `operation`. When the simulation terminates, it checks whether all clients were added to the set. When this is not the case, the liveness property is violated. We checked the two above properties for the above described ressource sharing system using SURTA

```
class MuTexVerification implements Verification
  var hadAccess as Set of CapsuleInstance = {}
  public function multipleAccess(rte as RunTimeEnvironment) as Boolean
    return exists client1 in rte.getCapsuleInstances("client"),
      client2 in rte.getCapsuleInstances("client") where client1 <> client2
      and client1.getCurrentState().getName = "operation"
      and client2.getCurrentState().getName = "operation"
  public function allHadAccess(rte as RunTimeEnvironment) as Boolean
    initially result as Boolean = true
    step
      if (forall controller in rte.getControllers() holds
        (not controller.executable()) and
        (not controller.containsNonEmptyBuffer())) then
        result := forall client in rte.getCapsuleInstances("client")
                    holds client in hadAccess
      else
        step foreach client in rte.getCapsuleInstances("client") where
               client.getCurrentState().getName() = "operation"
          add client to hadAccess
    step
      return result
  public procedure check(rte as RunTimeEnvironment)
    require (not multipleAccess(rte)) and allHadAccess(rte)
    skip
class RoseRTRunTimeEnvironment implements RunTimeEnvironment
  public procedure run(model as Model)
    step while (exists controller in controllers where (controller.executable()
                or controller.containsNonEmptyBuffer()))
      // Select an executable controller to execute.
      step
        try
          verification.check(me)
        catch
          e as Exception: WriteLine("Property violated.")
    step
      try // This additional check is for liveness properties.
        verification.check(me)
      catch
        e as Exception: WriteLine("Property violated.")
```

Fig. 14. A verification method to check mutual exclusion.

and SpecExplorer. The simulation revealed that the above mentioned liveness property was violated. To support debugging, we have SURTA report the state of the Rose-RT model after each interleaving step of the model execution, which results in an error trail when the property is violated. In our example, the output error trail suggests that the fairness property is violated because some client requests were removed when they occupied the head of the respective queue and could not be used at the moment. For more detail see [20].

7 Conclusion

We have presented an operational, executable semantics for a large portion of the syntactic features of the modeling language UML-RT. The semantics relies on a straightforward, syntactic translation of a UML-RT model into an AsmL representation. We use a layered architecture for the semantics definition, which interprets the syntactic representation using an AsmL-defined runtime layer.

The separation of syntactic representation and semantic interpretation greatly facilitates the implementation of semantic variation points, as well as the three-tiered approach to the different UML-RT semantics. We illustrate how to use the random walk simulation capability of SpecExplorer in order to show the violation of safety and liveness properties.

Future work includes an automatic translation of UML-RT models into the AsmL representation. We also work on providing a semantics for the syntactic features of UML-RT that we do not currently handle, a task that is facilitated by the flexible structure of our semantics definition. We will develop a methodology for model based testing based on SURTA. We also plan to generalize the simulation based property validation approach sketched in this paper, in particular by extending it to handle more general LTL properties. Finally, we expect a complete state space exploration for AsmL to become available, which would avail our semantics to complete model checking.

Acknowledgment. We thank Daniel Butnaru for his assistance in implementing the SURTA project.

References

1. K. B. Akhlaki, M. I. C. Tuñón, and J. A. H. Terriza. Design of real-time systems by systematic transformation of UML/RT models into simple timed process algebra system specifications. In *Proc. ICEIS (3)*, pages 290–297, 2006.
2. AsmL – Abstract State Machine Language (Microsoft). `http://research.microsoft.com/fse/asml`.
3. J. Bezerra and C. M. Hirata. A semantics for UML-RT using π-calculus. In *Proc. RSP 2007*, pages 75–82, 2007.
4. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
5. J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proc. WOSS*, pages 28–33. ACM, 2004.
6. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
7. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Technical Report MST-TR-2005-59, Microsoft Research, 2005.
8. M. I. Capel, L. E. M. Morales, K. B. Akhlaki, and J. A. H. Terriza. A semantic formalization of UML-RT models with CSP+T processes applicable to real-time systems verification. In *Proc. JISBD*, pages 283–292, 2006.
9. G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *ESEC / SIGSOFT FSE*, pages 186–195. ACM Press, 2001.
10. C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In *FASE 2001*, volume 2029 of *LNCS*. Springer Verlag, 2001.
11. M. Fuchs, D. Nazareth, D. Daniel, and B. Rumpe. BMW-ROOM: An object-oriented method for ASCET. In *SAE'98*. Society of Automotive Engineers, 1998.

12. Q. Gao, L.J. Brown, and L.F. Capretz. Extending UML-RT for control system modeling. *American Journal of Applied Sciences*, 1(4):338–347, 2004.
13. R. Grosu, M. Broy, B. Selic, and G. Stefanescu. Towards a calculus for UML-RT specifications. In *Proc. OOPSLA*, 1998.
14. G. Gullekson. *Designing for concurrency and distribution with Rational Rose RealTime*, 2003. Rational Software White Paper. `http://www.ibm.com/developerworks/rational/library/269.html`.
15. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
16. A. Habibi and S. Tahar. AsmL semantics in fixpoint. In *Proc. ASM*, pages 233–246, 2005.
17. D. Herzberg. UML-RT as a candidate for modeling embedded real-time systems in the telecommunication domain. In *UML'99*, volume 1723 of *LNCS*, pages 330–338. Springer, 1999.
18. M. Kardoš. *Automated formal verification for UML-based model driven design of embedded systems*. PhD thesis, Slovak University of Technology, 2006.
19. A. Knapp, S. Merz, and C. Rauh. Model checking timed UML state machines and collaborations. In *FTRTFT'02*, volume 2469 of *LNCS*, pages 395–416. Springer, 2002.
20. S. Leue, A. Ştefănescu, and W. Wei. An AsmL semantics for dynamic structures and run time schedulability in UML-RT. Technical Report soft-08-02, University of Konstanz, 2008. Available from `http://www.inf.uni-konstanz.de/soft/publications_en.php`.
21. OMG Model Driven Architecture (MDA). `http://www.omg.org/mda`.
22. R. Ramos, A. Sampaio, and A. Mota. A semantics for UML-RT active classes via mapping into Circus. In *FMOODS'05*, volume 3535 of *LNCS*, pages 99–114. Springer, 2005.
23. Rational Rose RealTime tool. Shipped within Rational Rose Technical Developer: `http://www.ibm.com/software/awdtools/developer/technical`.
24. M. Saaltink. Generating and analysing Promela from RoseRT models. Technical Report TR-99-5537-02, ORA Canada, 1208 One Nicholas Street, Ottawa Ontario, K1N 7B7, Canada, 1999.
25. M. Saksena, P. Freedman, and P. Rodzewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 240–25. IEEE Computer Society, 1997.
26. B. Selic. Turning clockwise: using UML in the real-time domain. *Comm. of the ACM*, 42(10):46–54, Oct. 1999.
27. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.
28. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.ibm.com/developerworks/rational/library/139.html`, March 1998.
29. Spec Explorer tool. `http://research.microsoft.com/SpecExplorer`.
30. M. von der Beeck. A formal semantics of UML-RT. In *Proc. MoDELS*, volume 4199 of *LNCS*, pages 768–782. Springer, 2006.