Technical Report soft-08-02, Chair for Software Engineering, University of Konstanz
DRAFT

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, actor importation, multiple containment, and substitutability. 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 model-based testing tool.

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

1 Introduction

UML-RT [34] was proposed as a UML dialect customized for the design of distributed embedded real-time systems [32]. UML-RT is based on the ROOM notation [33] which was originally developed at Bell Northern Research. The ROOM notation was first implemented in the ObjecTime Developer tool suite. Currently supported by the IBM Rational Rose RealTime (Rose-RT) tool [29], UML-RT finds applications in a broad range of domains including telecommunications [20], control systems [31, 15], and automotive systems [14]. A UML-RT model consists of a set of concurrent autonomous objects, called actors, that exchange messages with one another through dedicated communication interfaces

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

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, 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: Consider a Web-based service in which the server provides a separate session to serve each individual client. Since the maximal number of clients is unknown, it is both inflexible and resource inefficient to have a fixed number of sessions in the server. Instead, a more reasonable design would adopt dynamical creation and destruction of sessions.

Software models play a central role throughout the whole life cycle in development processes following the model driven architecture paradigm [24]. Models are used for documentation, prototyping, code generation and testing. It is therefore of great importance that a software model is correctly designed: the model must meet all the requirements of the software system. The only way to achieve full confidence in this correctness criterion is the use of formal methods, e.g., using systematic state space exploration. However, formal verification requires the availability of a formal semantics of the modeling language. Unfortunately, the semantics of UML-RT is only informally captured in [33], and there is little research invested into its formalization [16, 36, 23, 13, 12, 25, 11, 5]. In particular, there is no work, to the best of our knowledge, addressing important issues like dynamic structures and run time scheduling.

In this paper we report on the SURTA[2] 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 [4]. Developed by the Foundations of Software Engineering group at Microsoft Research [1], the AsmL language is supported by the Spec Explorer tool [35] whose most outstanding feature is the generation of a finite state machine from an AsmL program. This enables the simulation, assertion checking, and test case generation of the program by exploring the generated finite state machine. AsmL is also tightly integrated into the Microsoft .NET framework [2]. AsmL borrows powerful programming features from the .NET framework. In particular, we use the .NET type system for describing meta-model level details of UML-RT. Another 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 or only limited 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) run time incarnation and destruction of actors, (2) actor importation/deportation, (3) dynamic port binding, (4)

[2] SURTA stands for **S**emantics of **U**ML-**RT** in **A**smL.

replications, (5) transition guards/actions, (6) controllers, and (7) other run time environment features.
- The informal semantics of UML-RT as described in [33] has many ambiguities and semantic variation points whose decision choices depend on the special implementation in a vendor CASE tool. Our solution to this problem is to give a parametrized semantics for the unambiguous parts of UML-RT, which can then be extended with a concrete implementation of semantic variation points and interpretation of ambiguities.
- We implemented several variants of UML-RT semantics: (1) We define a most general semantics that can encompass all interpretations that are possible according to [33]; (2) We also define a semantics that is in line with the Communicating Finite State Machines (CFSMs) paradigm [9]; (3) We formalized the concrete semantics as the one 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. This gives a more convenient way of validating Rose-RT models.

The paper is structured as follows. First, we give a brief introduction to UML-RT and AsmL respectively in Sections 2 and 3. 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, and sketch a model-based testing approach based on the SURTA project in Section 7. In Section 8 we discuss related work, in Section 9 we suggest future work, and we conclude the paper in Section 10.

2 UML-RT

Using an example model as shown in Figure 1, we briefly introduce the set of UML-RT concepts and modeling elements 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 to return the computation result that adds 3 to a client-specified number. The request of a client prompts the client manager to send an unused service accessor object to the server. This object does nothing but forwards messages

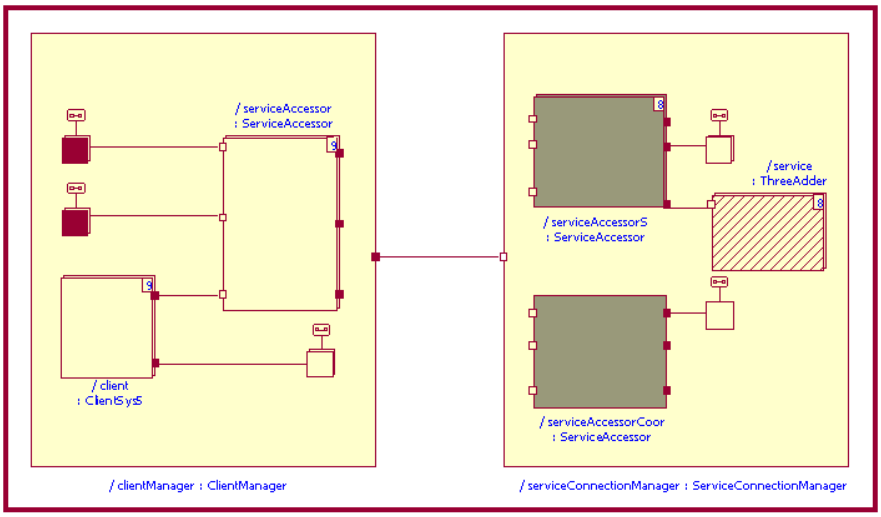

Fig. 1. A UML-RT model in which a number of clients request remote services. The model is taken from a collection of examples included in Rose-RT distributions, with slight modification to allow dynamic creation and destruction of `service` instances.

from one side of ports to another side of ports. The connection manager at the server 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.

Static structure. The central structural concept in UML-RT is that of an *actor*. 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 have a set of sub-actors. More precisely, an actor may hold a set of *actor references*, 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` of the class `ClientSys5`, `serviceAccessorS` of the class `ServiceAccessor`, etc. We say that 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 it is incarnated once the containing actor `clientManager` is created. On the contrary, 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` of the same class `ServiceAccessor`, 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. Note that a port is not a place to store incoming messages. Its role is to solely define a communication access point for other actors. 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 relay port is always located on the structural boundary of an actor, and allows messages to pass through the actor boundary to connect the inside and outside worlds of the actor. 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. When a replicated entity is not defined with a fixed finite replication factor, an unbounded number of members of the entity can be created at run time. Replication is used to obtain a more flexible and concise graphic representation of a model. For instance, instead of having nine actor references of the class `ClientSys5` inside the actor `clientManager`, one needs to only specify one replicated reference with the factor 9. Moreover, this allows the maximal number of clients to be conveniently changed by just modifying the replication factor of the reference, without the need to change the inner structure of `clientManager`. 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

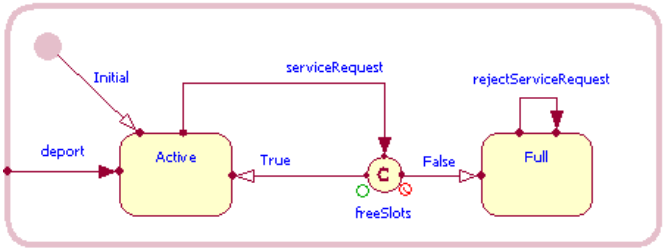

Fig. 2. The hierarchical state machine of the `serviceConnectionManager` in Figure 1

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.

```
const RTActorId & theClient = *rtdata;

frame.import(theClient, serviceAccessorCoor);
frame.import(theClient, serviceAccessorS);
connectionSetup.serviceReady().send();
++active;
frame.deport(theClient, serviceAccessorCoor);
```

Fig. 3. The C++ action code of the transition `serviceRequest` in Figure 2

Figure 2 shows the state machine of the service connection manager. When the manager is at the `active` state, it builds a connection with the client side when a client request is received. Figure 3 shows how such a connection is built, as specified in the action code of the transition `serviceRequest`: The first line extracts from the received message a pointer to an actor (`theClient`) that is then imported into two sub-actor references inside `serviceConnectionManager`. The imported actor will act as a messenger between the client and its assigned service object. Next, the client is informed of the successful connection by sending a message `serviceReady` through the port `connectionSetup`. This message will travel through the imported actor to arrive at the client side. After the client is informed, the actor reference `serviceAccessorCoor` is no longer needed and the imported actor is deported from the reference. However, the imported actor is still held by the other actor reference `serviceAccessorS` for passing service-related messages between the client and the server. The server allows only a certain number of requests to be accepted. So, the manager checks if the current number of connections reaches the maximum. When it is the case, it goes to the `full` state and rejects any further request. It returns to the `active` state when a service is completed and the corresponding connection is torn down (the

`deport` transition). In this paper we consider only flat state machines, and leave the formalization of hierarchical state machines for future work. Currently, we have to flatten hierarchical state machines, as illustrated in Figure 4. While the resulted state machine is still simple in our example, the flattening approach may result in an exponential blow-up of the size of a flattened state machine. Note that we also eliminate the choice point `freeSlots` in the original state machine. This choice point is to determine if the current number of connections exceeds the allowed maximum. In the flat state machine in Figure 4, we use two transitions `serviceRequest1` and `serviceRequest2` to replace the transition `serviceRequest`, and encode the choice point into the guards of the two new transitions: `serviceRequest1` has the guard condition that the current number of connections is smaller than the maximum, and `serviceRequest1` has the opposite condition. Furthermore, the two new transitions have the same action as seen in Figure 3.

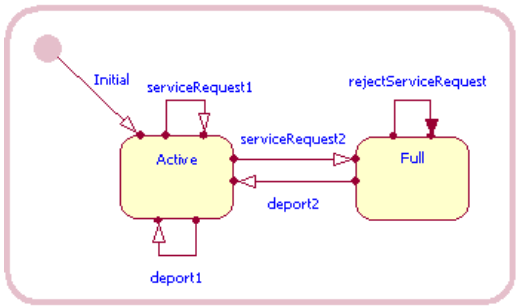

Fig. 4. The flattened state machine in Figure 2

Controllers. At run time, each actor instance runs on a thread. A thread may contain multiple actor instances, and the executions of these instances are interleaved in an order decided by the 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 modeling tool for distributed embedded systems [29, 17]. It supports the UML-RT formalism and provides a full cycle software development from high-level design, testing, to automatic code generation. Rose-RT currently supports three programming languages: C [26], C++ [27], and Java [28], for specifying transition actions in state machines.

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 5. The replicated

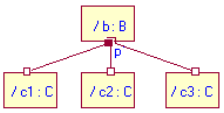

Fig. 5. A UML-RT model.

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 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 [33].

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 [18] 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 6. 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.

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*. On the contrary, the same checking in the next step returns *true* since at that moment the interval is already updated.

```
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. 6. An AsmL program of integer intervals.

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. The constraint in the class *Interval* says that the lower bound of an interval should be no larger than the upper bound. Assertion checking does not only help revealing program errors, it also enables property verification by searching for assertion violations, as explained in Section 6.

The AsmL language is based on the Abstract State Machine formalism [7]. A formal semantics has been defined for the core of AsmL [18]. There are also other semantic definitions for AsmL such as the one using the theory of fixpoints [19]. Even though 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. This section gives an overview of the ongoing project. We argue that our choice of specification architecture offers great flexibility for implementing different semantic variants, and simplifies the transformation of a concrete UML-RT model to its AsmL specification.

Note that our objective is to give a semantics for the UML-RT language. We do not consider here 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., [25, 3, 5]. In the SURTA project, we separate the syntactic elements from the semantics of run time entities. With such an approach, the transformation of a UML-RT model can 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 common semantics of a chosen run time environment. We argue that 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. This advantage was already observed when we implemented different concrete semantics in the SURTA project.
– Because a concrete UML-RT model is only syntactically transformed into AsmL, a change in the semantic level does not need 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.

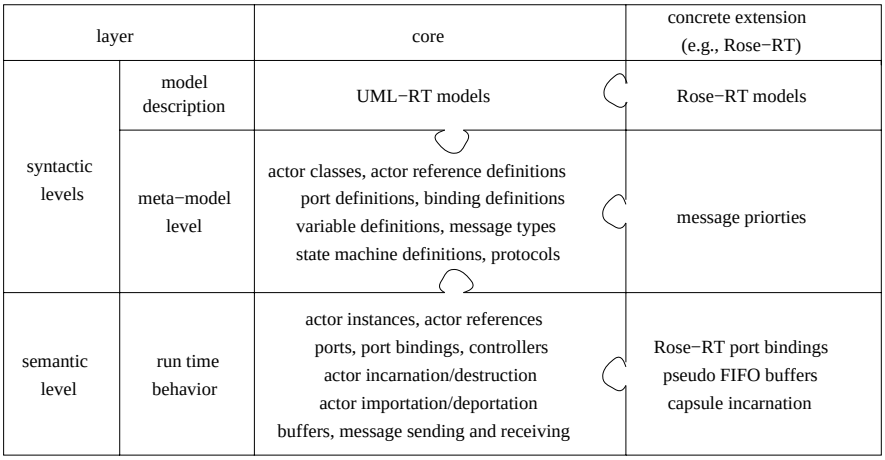

Fig. 7. The SURTA architecture.

Figure 7 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 and interpretations 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, e.g., different implementations of port bindings, message passing behavior, controllers, etc. 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) [9], 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 all three concrete extensions will be presented with details in Section 5 as well as the core of the SURTA project. Before detailed discussions, Table 1 summarizes all semantic variation points that we consider and how they are differently realized in the three extensions mentioned above.

Variation point	Most general semantics	CFSM-based semantics	Rose-RT semantics
Port binding	Total non-determinism	Total non-determinism	Using priorities based on replicated entity indices
Message buffer data structure	Bag-like data structures	FIFO queues	FIFO queues
Message buffer allocation	One buffer per end port	One buffer per end port	Buffers are shared by all ports of all actors in one controller
Assignment of actors to controllers	One actor per controller	One actor per controller	An actor and all its fixed sub-actors reside in one same controller

Table 1. Semantic variation points and their realizations.

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. We present one or several closely-related UML-RT concepts in each subsection and, when encountering a semantic variation point, also discuss its different realizations in the three concrete extensions mentioned in the end of Section 4.

5.1 Actor Classes

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 8. The constraint *distinctSubActorRefNames* in the class definition says that no two sub-actor references share the same name inside an actor. This constraint is necessary since when incarnating an actor reference, the reference is located by its name inside the enclosing actor. We also specify a number of constraints ensuring distinct port names, the well-formedness of binding definitions and state machines, etc. 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 5.6.

Actor reference definitions. In the class of sub-actor reference definitions, the field *kind* specifies whether the reference is fixed, optional, or imported. It also contains a replication factor which is enforced to be a positive number by the constraint *postiveReplicationFactor*. This restriction is however not given in the informal UML RT semantics in [33]. Nevertheless, it makes no sense to have a negative number of actor references. In Rose-RT, when an actor reference (called capsule role) is defined to have a negative replication factor, a complaint will be issued at compile time, and the negative factor is automatically reset to 0. In this case, the corresponding reference is not allowed to contain any pointers, i.e., it cannot be incarnated or host an imported actor.

```
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
  // We omit class methods and constraints except the following.
  constraint distinctSubActorRefNames:
    forall rDef1 in subActorRefDefs, rDef2 in subActorRefDefs
      where rDef1 <> rDef2 holds
        rDef1.getName() <> rDef2.getName()

class SubActorRefDefinition
  private const name as String
  private const myClass as ActorClass
  private const kind as ActorReferenceKind
  private const replicationFactor as Integer
  // We omit class methods.
  constraint positiveReplicationFactor:
    replicationFactor > 0

class PortDefinition
  private const name as String
  private const kind as PortKind
  private const myProtocol as Protocol
  private const replicationFactor as Integer
  // We omit class methods and constraints.

class BindingDefinition
  private const actor1 as SubActorRefDefinition?
  private const port1 as PortDefinition
  private const actor2 as SubActorRefDefinition?
  private const port2 as PortDefinition
  // We omit class methods and constraints except the following:
  constraint boundPortsCompatible:
    // We omit the implementation of the constraint.

class VariableDefinition
  private const name as String
  private const dataType as Type
  // We omit class methods and constraints.
```

Fig. 8. The AsmL definition of actor classes.

Port definitions. A port definition has one of three types. It is 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.

Binding definitions. There is a trick in the class of binding definitions. In Figure 1, the internal end ports of the class *ClientManager* are connected to some ports of its sub-actor reference *serviceAccessor*. This results in circular definitions: in order to define the class *ClientManager*, one has to provide the definition of the bindings that, in our case, rely on the definition of the *ClientManager*. We break this undesirable circle by allowing the *actor1* and *actor2* fields to contain a *null* value[3]. Whenever a sub-actor reference in a binding definition is *null*, we know that the respective port belongs to the actor class that contains the binding definition. The class of binding definitions are specified with a set of constraints to guarantee protocol compatibility and port visibility.

Rose-RT syntactic variants. Actors are called *capsules* in Rose-RT. To adopt the Rose-RT terminology, we create several name aliases: e.g., *CapsuleClass* for *ActorClass*, *FixedCapsuleRoleDefinition* for *SubActorRefDefinition* whose kind is *fixed*, *ConnectorDefinition* for *BindingDefinition*, etc. They are created just to enhance the readability of a translated Rose-RT model.

5.2 State Machines

As mentioned before, we formalize here only flat state machines. Hierarchical state machines are left for future work.

State machines. The class of state machines is defined in Figure 9, consisting of a set of states, transitions, and an initial state. The initial state of a state machine must be contained in the set of the states of the machine, as required by the constraint *validInitialState*. A transition has a source state, a target state, one or more triggers, and an optional action.

Triggers. 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.

Guards. We define transition guards as an AsmL interface with a mandatory *evaluate* method that returns a Boolean value. The *evaluate* method is to be implemented for each individual 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.

[3] The type *SubActorRefDefinition?* is a shorthand of *SubActorRefDefinition or Null* where *Null* is a special type containing only one distinct value *null*. The value *null* is often used to indicate an uninitialized value.

```
class StateMachine
  private const states as Set of State
  private const initialState as State
  private const transitions as Set of Transition
  // We omit class methods and constraints except the following:
  constraint validInitialState: initialState in states

class State
  private const name as String
  private const entryAction as Action?
  private const exitAction as Action?
  // We omit class methods and constraints.

class Transition
  private const name as String
  private const source as State
  private const target as State
  private const triggers as Set of Trigger
  private const action as Action?
  // We omit class methods and constraints.

class Trigger
  private const signal as String
  private const port as EndPortDefinition
  private const guard as Guard?
  // We omit class methods and constraints.

interface Guard
  evaluate(actor as ActorInstance, message as Message) as Boolean

interface Action
  execute(actor as ActorInstance, re as RunTimeEnvironment) as Boolean
```

Fig. 9. The AsmL definition of state machines.

Actions. Like guards, actions are defined as an interface. An implementation of the mandatory *execute* method specifies the concrete action to be executed. Unlike guards, the method 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.

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.

5.3 Protocol

As illustrated in Figure 10, 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, as seen in the *conjugate* method of the *Protocol* class.

```
class Protocol
  private const name as String
  private const incomings as Set of MessageType
  private const outgoings as Set of MessageType
  // We omit class methods and constraints except the following:
  public function conjugate() as Protocol
    return new Protocol(name + "/Conjugated", outgoings, incomings)

structure MessageType
  signal as String
  dataType as Type

class Message
  private const signal as String
  private const priority as Integer
  private const data as Obj
  // We omit class methods and constraints except the following:
  public function ofMessageType(aType as MessageType) as Boolean
    return (signal, data.GetType()) = (aType.signal, aType.dataType)
```

Fig. 10. The AsmL definition of protocols.

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. This is stipulated as the *ofMessageType* function in the *Message* class.

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.

Until now we have been concerned with the meta-model level mappings. Starting with the next subsection we will discuss the run time behavior level that gives semantics for run time UML-RT entities.

5.4 Actor References

Figure 11 defines the class of actor references. Note that it contains merely 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. The *setInstanceAt* method adds an actor pointer to an unincarnated reference, and this pointer can also be removed by calling *removeInstanceAt*. These two methods are needed for dynamic structures, e.g., incarnating or importing an actor.

```
class ActorReference
  private const myDef as SubActorRefDefinition
  private var instance as Map of Integer to ActorInstance
  // We omit class methods and constraints except the following
  public function isInstantiatedAt(index as Integer) as Boolean
    require index in {0..getReplicationFactor() - 1}
    return index in instance
  public procedure setInstanceAt(index as Integer, inst as ActorInstance)
    require (not isInstantiatedAt(index)) and inst.getClass() = getClass()
    instance(index) := inst
  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. 11. The AsmL definition of actor references.

5.5 Ports

The class *Port* defines actual ports created at run time, as shown in Figure 12, 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.

5.6 Actor Instances

Fields. Figure 13 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

```
class Port
  private const myDef as PortDefinition
  // We omit class methods and constraints.

class EndPort extends Port
  private var peer as Map of Integer to (Port, Integer) = {->}
  // We omit class methods and constraints except the following.
  public function isBoundAt(index as Integer) as Boolean
    require index in {0..getReplicationFactor() - 1}
    return index in peer
  public procedure setPeerAt(index as Integer, p as Port, pIndex as Integer)
    require not isBoundAt(index)
    peer(index) := (p, pIndex)

class RelayPort extends Port
  private var peer as Map of (Integer, Boolean) to (Port, Integer) = {->}
  // We omit class methods and constraints except the following.
  public function isBoundAt(index as Integer, side as Boolean) as Boolean
    require index in {0..getReplicationFactor() - 1}
    return (index, side) in peer
  public procedure setPeerAt(index as Integer, side as Boolean,
                             p as Port, pIndex as Integer)
    require not isBoundAt(index, side)
    peer(index, side) := (p, pIndex)
```

Fig. 12. The AsmL definition of run time ports.

state of the actor instance. We also use an auxiliary field *bindingsForImportedRef* to remember which run time port bindings were established when an actor was imported to a particular sub-actor reference. This information is needed when the actor is deported, and the bindings constructed for this actor at the importation time need to be torn down.

```
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
  private var bindingsForImportedRef
    as Map of (ImportedActorReference, Integer) to (Set of Binding) = {->}
```

Fig. 13. The AsmL definition of actor instances.

Constructor and port binding. Figure 14 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 constructor method of *ActorReference* (Figure 11) makes sure that all fixed sub-actor references are incarnated. The most intricate part of actor instance construction is how ports inside should be bound, which is very vaguely described in [33]. This is especially the case 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 14 shows the most general binding strategy: when a port p can be bound to either p_1 or p_2, we nondeterministically choose one of p_1 and p_2 to be bound with p. This semantics is used in the most general extension and in the CFSM-based extension that we have implemented in SURTA. On the contrary, the extension based on Rose-RT uses a different port binding semantics as described as follows.

```
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()}
    initially possibleBindings as Set of Binding = getAllPossibleBindings()
    step while exists binding in possibleBindings
      where binding.bothPartiesFree()
      choose binding in {b | b in possibleBindings where b.bothPartiesFree()}
      binding.bind()
      remove binding from possibleBindings
```

Fig. 14. The constructor method of the *ActorInstance* class.

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.

- 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.
- 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.
- 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.
- 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, as shown in Figure 15. The class *CapsuleInstance* extends the class *ActorInstance*. The method *activate* in the constructor binds ports as follows. For each binding definition of the respective capsule class, the method checks if there are available ports to be bound. When this is the case, for each party involved in the binding definition, it calls the *getAvailablePort* method that first locates the actor reference with the minimal index that has free ports, and subsequently returns the free port with the minimal index in the reference. 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 Rose RT implementation of port binding here.

Actor reference incarnation and destruction. The method *incarnateAt* of the class *ActorInstance*, as shown in Figure 16, shows how a sub-actor reference is incarnated at a particular index. The method restricts the incarnated reference to be optional, because fixed references must have been incarnated when the containing actor was created, and imported references cannot be incarnated. This restriction is expressed by requiring the parameter *aRef* to have the type *OptionalActorReference*, which is a subtype of *ActorReference*. The method first checks whether the reference is already incarnated at the index *index*. If not, a new actor instance of the respective class is created and the reference *aRef* at *index* obtains a pointer to the newly created actor. Next, 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 [33] 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 17, 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

```
class CapsuleInstance extends ActorInstance
    mybase(aClass, ActorInstanceKind.RoseRT)

    step foreach bDef in aClass.getBindingDefs()
      activate(bDef, ports, subActorRefs)

procedure activate(bDef as BindingDefinition,
                   ports as Set of Port,
                   refs as Set of ActorReference)
  // ... ...
  step while (hasAvailablePorts(bDef, 1, ref1, ports)
    and hasAvailablePorts(bDef, 2, ref2, ports))
    (port1, index1) = getAvailablePort(bDef, 1, ref1, ports)
    (port2, index2) = getAvailablePort(bDef, 2, ref2, ports)
    let binding = new Binding(port1, index1, side1, port2, index2, side2)
    binding.bind()

function getAvailablePort(bDef as BindingDefinition, party as Integer,
                          aRef as ActorReference?, ports as Set of Port)
   as (Port, Integer) or Null
   initially result as (Port, Integer) or Null = null
   step
     // ... ...
      let index = min i | i in {0..aRef.getReplicationFactor() - 1} where
        hasAvailablePorts(bDef, party, aRef, i)
      // ... ...
        let actor = aRef.deref(index)
        let port = the p | p in actor.getPorts() where p.getDef()
          = bDef.getPortDef(party)
        let mapi = getMinimalAvailablePortIndex(port, true)
        if (mapi <> -1) then
          result := (port, mapi)
     // ... ...
   step
     return result
```

Fig. 15. The incarnation method of the *CapsuleInstance* class.

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.

Rose-RT capsule instances. The implementation of the Rose-RT capsule instances differs only with respect to the port binding strategy as described above.

```
public procedure incarnateAt(aRef as OptionalActorReference,
                             index as Integer)
  require aRef in subActorRefs and not aRef.isInstantiatedAt(index)
  step
    aRef.setInstanceAt(index, new ActorInstance(aRef.getClass()))
  step
    // We omit here port binding using the most general strategy.

public procedure destroyAt(aRef as OptionalActorReference,
                           index as Integer)
  require aRef in subActorRefs and aRef.isInstantiatedAt(index)
  step foreach port in aRef.deref(index).getPorts()
    unplug(port)
  step
    aRef.removeInstanceAt(index)
```

Fig. 16. The AsmL implementation of actor reference incarnation and destruction.

5.7 Controllers

We define an interface for controllers as in Figure 18, which provides the signatures for a set of basic controller operations. Various concrete controllers can be implemented using this interface. 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.

The most general controllers. Such a controller can host only one actor instance, and create a separate bag-like message buffer for each end port of the actor. Any message in a message buffer can be used to trigger a transition. The order of messages in a buffer is therefore irrelevant. In each step of the execution of a controller, it checks the set of outgoing transitions of the current state of the hosted actor to see if any transition is fireable by some message in all buffers. If there is any fireable transition, the controller randomly picks one to fire.

CFSM-based controllers. The behavior of a CFSM-based controller is the same as the one of the most general controllers, except that FIFO message queues are instead used to store incoming messages for each end port of the hosted actor. Only the head message of a buffer can be used to trigger a transition.

Rose-RT controllers. Figure 19 gives the fields of the class of 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. The

```
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
      initially possibleBindings as Set of Binding =
        getAllPossibleBindings(..)
      initially activatedBindings as Set of Binding = {}
      step while exists binding in possibleBindings
        where binding.bothPartiesFree()
        // bind port here.
        // We omit the details of recording established bindings
        // in bindingsForImportedRef(aRef, index).
    else
      WriteLine("No binding possible. Importation fails.")
      throw new Exception()

public procedure deportAt(aRef as ImportedActorReference,
                          index as Integer)
  require aRef in subActorRefs and aRef.isInstantiatedAt(index)
  step foreach binding in bindingsForImportedRef(aRef, index)
    binding.unbind()
  step
    bindingsForImportedRef :=
      {i -> bindingsForImportedRef(i) |
         i in bindingsForImportedRef where i <> (aRef, index)}
  step
    aRef.removeInstanceAt(index)
```

Fig. 17. The AsmL implementation of actor importation and deportation.

```
interface Controller
  getActors() as Set of ActorInstance
  contains(actor as ActorInstance) as Boolean
  getLastReceivingPort(actor as ActorInstance) as Port?
  getLastReceivingPortIndex(actor as ActorInstance) as Integer
  getLastReceivedMessage(actor as ActorInstance) as Message?
  executable() as Boolean
  makeStep(re as RunTimeEnvironment)
```

Fig. 18. The AsmL interface of controllers.

RoseRTController employs the field *messageToPort* to remember to which port a message will be delivered. Like other types of controllers, it also remembers the last port through which a hosted actor receives messages and the last message that an actor received. This information is needed by the run time environment to evaluate transition guards and to send reply messages.

```
class RoseRTController implements Controller
  private var actors as Set of ActorInstance
  private const internalBuffer
    as Map of RoseRTMessagePriority to FIFOMessageBuffer
    = {mp -> new FIFOMessageBuffer() | mp in enum of RoseRTMessagePriority}
  private const incomingBuffer
    as Map of RoseRTMessagePriority to FIFOMessageBuffer
    = {mp -> new FIFOMessageBuffer() | mp in enum of RoseRTMessagePriority}
  var messageToPort as Map of Message to (EndPort, Integer) = {->}
  private var lastReceivingPort as Map of ActorInstance to Port
  private var lastReceivingPortIndex as Map of ActorInstance to Integer
  private var lastReceivedMessage as Map of ActorInstance to Message
```

Fig. 19. The AsmL implementation of Rose-RT controllers.

Scheduling executions of hosted actors by Rose-RT controllers. The *makeStep* method of the class *RoseRTController*, as shown in Figure 20, 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. This is checked by the *executable* function that simply checks if there is a fireable transition by calling the *getFireableTransitions* function. 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. The chosen transition is fired by executing the exit action of the current state, executing the transition action, moving to the next state (*move*), and executing the entry action of the new current state.

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 sending. Figure 21 shows a part of the AsmL interface for run time environments. The last method *run* is used to execute a model. The other methods

```
public procedure makeStep(re as RunTimeEnvironment)
  require executable()
  choose (actor, transition, buffer) in getFireableTransitions()
    step
      // Receive message here.
    step
      if actor.getCurrentState().hasExitAction() then
        actor.getCurrentState().getExitAction().execute(actor, re)
    step
      if transition.hasAction() then
        let result = transition.getAction().execute(actor, re)
    step
      actor.move(transition)
    step
      if actor.getCurrentState().hasEntryAction() then
        actor.getCurrentState().getEntryAction().execute(actor, re)

function getFireableTransitions()
  as Set of (ActorInstance, Transition, FIFOMessageBuffer?)
  return
    searchFireableTransitions(true, RoseRTMessagePriority.synchronous)
```

Fig. 20. The AsmL implementation of the Rose RT controller scheduling method.

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
  getLastReceivingPort(actor as ActorInstance) as Port
  getLastReceivingPortIndex(actor as ActorInstance) as Integer
  getLastReceivedMessage(actor as ActorInstance) as Message
  getActorInstance(actor as ActorInstance, refName as String, index as Integer)
    as ActorInstance
  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. 21. The AsmL interface of run time environments.

The most general run time environment. In the most general run time environment, a new controller is created for each newly created actor instance. Con-

trollers run on separate threads in any run time environment, and the order of controller executions is therefore non-deterministic. In each interleaving step of executing a model, the most general run time environment checks which controllers are executable and randomly chooses one to make a step.

CFSM-based run time environment. The CFSM-based run time environment also creates a new controller for each actor, and non-deterministically chooses one executable controller to make a step.

Rose-RT run time environment. Unlike the other two run time environments, an actor and all its fixed sub-actors are hosted in one controller. Only when an optional sub-actor reference is to be incarnated, users are given an option if a new controller should be created for hosting the new incarnated actor. Figure 22 shows the implementation of the *run* method in the class *RoseRTRunTimeEnvironment*. 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. However, this may cause the loss of messages that are later needed. Alternatively, Rose-RT may employ a message deferral/recall mechanism: A user-defined action may choose to delay a certain message, which will then be stored in a special queue in the corresponding controller. This message can be made available later for the receiver actor. We currently do not implement message deferral in SURTA, and leave this aspect of the semantics for future work.

5.9 Model Descriptions

The AsmL specification of a concrete UML-RT model can be represented simply by a set of syntactic definitions of model elements based on the meta-model level data structures.

The general concept of a model is defined as an interface in Figure 23. The particular Rose-RT implementation of the interface is composed of a set of capsule classes and the top capsule class. The set of protocols is not explicitly presented.

The SURTA project uses a global method *getModel* to construct the AsmL specification of a Rose-RT model. Users specify the syntactic definition of a model inside the body of this method. 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. Figure 24 gives partly the AsmL specification of

```
class RoseRTRunTimeEnvironment implements RunTimeEnvironment
  private var controllers as Set of RoseRTController
  // We omit class methods and constraints except the following.
  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. 22. The AsmL implementation of the Rose-RT run time environment.

```
interface Model
  function getName() as String
  function getActorClasses() as Set of ActorClass
  function getTopActorClass() as ActorClass
  function report()

class RoseRTModel implements Model
  private const name as String
  private const capsuleClasses as Set of CapsuleClass
  private const topCapsuleClass as CapsuleClass
  // We omit class methods and constraints.
```

Fig. 23. The AsmL interface of UML-RT models.

the model in Figure 1. Note that the class *ConnectorDefinition* is just a synonym for *BindingDefinition*.

Special treatment is needed for defining transition guards and actions. Recall that concrete guards and actions are defined as special implementations of the respective interface. For instance, consider the transition *SCMSTRTran* that corresponds to the `serviceRequest1` transition in Figure 4. It has an action object of the class *SCMSRAction* as shown in Figure 25, which is a straightforward translation from the C++ code in Figure 3. Due to the different expressiveness and semantics of Java or C++ and AsmL, we cannot map the whole Java or C++ language into AsmL. However, a large part of Java and C++ features and

```
getModel() as Model
  // We omit the definitions of message types except the following.
  let serviceRequest = MessageType("serviceRequest", type of ActorInstance)
  // Define the ServiceAccessor class. Details omitted.
  // Define the ClientSys5 class. Details omitted.
  // Define the ThreeAdder class. Details omitted.

  // Define the ServiceConnectionManager class
  // We omit the definitions of port and binding definitions except the following.
  let serviceAPort =
    new ExternalWiredEndPortDefinition("serviceAccess", serviceAccessProt, true)
  let connectionSPort =
    new InternalWiredEndPortDefinition("connectionSetup", connectionSetupProt, true)
  let service = new FixedCapsuleRoleDefinition("service", threeAdderClass, 8)
  let serviceAccessorS =
    new ImportedCapsuleRoleDefinition("serviceAccessorS", serviceAccessorClass, 8)
  let serviceAccessorCoor =
    new ImportedCapsuleRoleDefinition("serviceAccessorCoor", serviceAccessorClass)
  let SCMConn1 =
    new ConnectorDefinition(serviceAccessorS, connectionTerminationSPort, connectionTPort)
  // We omit the definitions of states and transitions except the following.
  let SCMActive = new State("Active")
  let SCMFull = new State("Full")
  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)

  // Define the top container class System5.
  let clientManager = new FixedCapsuleRoleDefinition("clientManager", clientManagerClass)
  let serviceConnectionManager = new FixedCapsuleRoleDefinition(
    "serviceConnectionManager", serviceConnectionManagerClass)
  let System5Conn = new ConnectorDefinition(
    clientManager, serviceAccessPort, serviceConnectionManager, serviceAPort)
  let System5Class =
    new CapsuleClass("System5", {clientManager, serviceConnectionManager},
                     {}, {System5Conn}, {})

  let capsuleClasses = {System5Class, serviceAccessorClass, clientSys5Class,
                        clientManagerClass, threeAdderClass,
                        serviceConnectionManagerClass}
  let model = new RoseRTModel("System5", capsuleClasses, System5Class)
  return model
```

Fig. 24. The AsmL specification of the model in Figure 1.

statements can still be straightforwardly and automatically translated to AsmL equivalents.

```
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
      actor.setValue("active", actor.getValue("active") as Integer + 1)
    step
      re.deportActor(actor, "serviceAccessorCoor")
    step
      return true
```

Fig. 25. The AsmL implementation of the action of the transition *SCMSRTTran* in Figure 24.

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 [10]. A potential of the use of SURTA in model-based testing is sketched in Section 7. Model checking is impeded by the fact that the currently publicly available version of Spec Explorer is not completely exploring the state space, in particular if the AsmL model includes non-deterministic choices, as is the case in SURTA code. We hence illustrate how the random walk simulation feature of Spec Explorer can be used to reveal property violations for Rose-RT models.

We first examine how the violation of a property can be discovered using the simulation capabilities of the Rose-RT tool. User can check the state machine execution of each capsule instance, the run time value of a variable, and the content of a message buffer during a simulation. However, such an approach can become very inconvenient. As an example, consider a simple model of mutual exclusion in which a central server grants exclusive access to a shared object to multiple clients. In the model, a client is possessing the object access when it

is in the `operation` state. One important property is that no more than one client is allowed to have the access at any time, i.e., only one client can be at the `operation` state. In order to check this property with Rose-RT, one has to carry out a step-by-step simulation of the model. In each step, the user has to check the state of each client and see if more than one client is at `operation`. Imagine that there are hundreds of clients in the model, and how tedious this checking can be.

The checking of the above mentioned property can be easily done with SURTA: We encode the negation of the property into the function *multipleAccess* in a *MuTexVerification* class, as shown in Figure 26. 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.

The above property is a safety property, i.e., a property that nothing bad will ever happen. We may also easily encode some liveness properties, i.e., properties that something good will eventually happen. One example is that each client will eventually have the object access, as encoded in the *allHadAccess* method in Figure 26. 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 if 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 a Rose RT model of mutual exclusion in the SURTA project. 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 as shown in Figure 27 suggests that the fairness property is violated because some client request was removed (Line 16) when it occupied the head of the respective queue and could not be used at the moment.

In principle any LTL property can be encoded and checked using the above technique in the SURTA project. This is because any LTL property can be transformed into a finite automaton that can be encoded as AsmL methods. In future work we will automate the translation of LTL properties into AsmL.

The above approach cannot prove the satisfaction of a property, and can only reveal property violations. As mentioned above, one cannot use the feature of finite state machine generation in Spec Explorer either to check that a property is satisfied: Spec Explorer does not generate the complete global state space of a model when non-deterministic statements, such as `any` and `choose`, are used in the model. This is because Spec Explorer does not exhaustively all the outcomes of executing a non-deterministic statement. This is intended to reduce the overhead of state machine generation, and it is encouraged to encode non-deterministic choices using method parameters. However, such encoding has difficulties to deal with nested and recursive choices that are heavily used in SURTA.

```
interface Verification
  check(rte as RunTimeEnvironment) as Boolean

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

verification = new MuTexVerification()

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. 26. A verification method to check mutual exclusion.

```
1  A new capsule instance of "Server" created: ci1
2  A new capsule instance of "MuTex" created: ci0
3  ci1(Server) enters State "Idle"
4  A new instance of "Client" created: gai0
5  A new instance of "Client" created: gai1
6  ci0(MuTex) enters State "Running"
7  gai0 update the local variable "processID" to value "1"
8  A message "request" sent at "cPort"(gai0)
9  gai0(Client) enters State "Waiting"
10 gai1 update the local variable "processID" to value "1"
11 A message "request" sent at "cPort"(gai1)
12 gai1(Client) enters State "Waiting"
13 A message "request" received at "sPort1"(ci1)
14 A message "reply" sent at "sPort1"(ci1)
15 ci1(Server) enters State "Waiting"
16 A message "request" was removed without being dispatched to its receiver.
17 A message "reply" received at "cPort"(gai0)
18 A message "release" sent at "cPort"(gai0)
19 gai0 update the local variable "processID" to value "2"
20 gai0(Client) enters State "Done"
21 A message "release" received at "sPort1"(ci1)
22 ci1(Server) enters State "Idle"
23 PROPERTY VIOLATION: Some client has not entered the critical region.
24 The execution of the model terminates.
```

Fig. 27. An error trail reported by SURTA.

An AsmL model checker, called *AsmLMC*, can be used to check the whole class of CTL* properties for AsmL programs [21]. However, it uses the exploration engine of Spec Explorer, and cannot exhaustively explore a model in the presence of non-deterministic choices either.

7 Model-Based Testing with SURTA

Rose-RT provides virtually no direct support for test case generations from Rose-RT models. We suggest how SURTA can be helpful in model-based testing, which still needs practical evaluations in future work.

A model-based testing approach with Spec Explorer consists of the following steps [10]: (1) A state machine model is generated from a given AsmL specification. The complete state space of the specification may be infinite or too large, in which case different techniques can be used to discard some part of the complete state space [10]. The resulting state machine is a test suite. (2) The actions in the AsmL specification are mapped to the procedures and functions of the implementation under test (IUT). (3) The implementation is executed, guided by the potential sequences of actions present in the state machine model. During the execution, the state of the AsmL specification and the state of the IUT are compared for revealing implementation errors.

SURTA may bring great convenience for mapping AsmL actions to procedures of an IUT when the IUT is synthesized from a model using the code generation feature of Rose-RT. This is because the code synthesized by Rose-RT preserves the state/transition structure of each capsule, which can be directly mapped to their equivalent in the AsmL syntactic definition of the Rose-RT model.

8 Related Work

Compared to a rich collection of formalization works for UML, there has been much fewer attempts to assign formal semantics to UML-RT, most of which are discussed below.

[16] is among the earliest efforts of formalizing UML-RT. It 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 shown how such a transformation can benefit the analysis of UML-RT models.

[36] uses labeled transition systems to give a formal semantics for a subset of UML-RT mostly focusing on the behavior of state machines. It also addresses hierarchical structures, but does not consider any dynamic changes in structures.

[23] covers the timed aspects of UML-RT by translating a timed state machine into a timed automaton. The aim of the work is to achieve a model checking approach for timed UML-RT models, and therefore it does not consider the semantics of any other aspects of UML-RT.

There are several approaches of formalizing UML-RT using process algebras [13, 12, 25, 11, 5]. Most of these works consider only synchronous communication. [12] considers asynchronous communication, but it allows only the use of *bounded* FIFO message buffers.

[5] gives semantics to the so-called *unwired* ports using name passing in π-calculus. The connectivity of an unwired port is not defined in the syntactic description of a model, and its binding to other unwired ports is completely controlled by user-defined actions. The support of unwired ports enables changes in the communication topologies. However, it does not consider other kinds of dynamic structures that our work is addressing.

All the above semantics works have the following common disadvantages: (1) None of them allows dynamic creation and destruction of actors as well as multiple containment. (2) They do not address 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.

Other related works include [6] that does not consider the formalization of UML-RT. Instead, it translates UML+, a dialect of UML RT whose semantics has been formally defined, to UML-RT in order to take advantage of the powerful CASE tool supports for UML-RT.

[22] 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.

An early approach to model checking RoseRT models is described in [30]. 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.

[8] 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.

9 Future Work

We are working towards a complete semantics that covers all aspects of UML-RT. There are currently some missing ingredients, such as synchronous communication, unwired ports, hierarchical state machines, just due to time constraints. Thanks to the expressiveness of AsmL, the semantic specifications of these features can be straightforwardly done.

- **Synchronous communication.** After an actor a_1 initiates a synchronous communication with another actor a_2, the run time environment will stop choosing a_1 for execution until a_2 takes part in the communication that a_1 initiates.
- **Unwired ports.** A port in SURTA uses a *peer* field to record its communication partner. An unwired port can be bound to any free compatible unwired port by just setting the *peer* field. So, the implementation of dynamically binding an unwired port is not a problem. The main challenge lies in the support of the registering, publishing, and discovering of unwired ports by the run time environment.
- **Hierarchical state machines.** UML-RT state machines do not support concurrent regions inside a composite state. We need only to address the concepts of history and group transitions. We may borrow the ideas of the specification of hierarchical state machines in [22].

In another important direction of future work, we will integrate our SURTA project into real life software development practices, providing supports for model validation and model-based testing. We are currently developing an automated tool that transforms a Rose-RT model into its syntactic description in AsmL. We will also develop an automated procedure to encode LTL properties in AsmL. Finally, we will evaluate how SURTA can better serve model-based testing by providing direct mappings between Rose-RT models and their implementations generated by the Rose-RT tool.

10 Conclusion

We have presented an operational, executable semantics for a large portion of the syntactic features of the modeling language UML-RT. We use a layered architecture for the semantics definition, which interprets the syntactic representation using an AsmL-defined run time 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. Using this architecture we are able to specify semantics for dynamic structures, replications, port bindings, run time schedulability and other UML-RT features that no previous work has covered. The availability of the semantics for these features enables a broader range of realistic system models to be formally analyzed. We illustrate how to use the random walk simulation capability of Spec Explorer in order to show the violation of safety and liveness properties.

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

References

1. Foundations of software engineering group. `http://research.microsoft.com/foundations/`.
2. Microsoft .NET. `http://www.microsoft.com/net/`.
3. 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.
4. AsmL – Abstract State Machine Language (Microsoft). `http://research.microsoft.com/fse/asml`.
5. J. Bezerra and C. M. Hirata. A semantics for UML-RT using π-calculus. In *Proc. RSP 2007*, pages 75–82, 2007.
6. V. Del Bianco, L. Lavazza, M. Mauri, and G. Occorso. Towards UML-based formal specifications of component-based real-time software. *STTT*, 9(2):179–192, 2007.
7. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
8. 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.
9. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
10. 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.
11. 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.
12. 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.

13. 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.
14. 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.
15. 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.
16. R. Grosu, M. Broy, B. Selic, and G. Stefanescu. Towards a calculus for UML-RT specifications. In *Proc. OOPSLA*, 1998.
17. 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`.
18. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
19. A. Habibi and S. Tahar. AsmL semantics in fixpoint. In *Proc. ASM*, pages 233–246, 2005.
20. 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.
21. M. Kardoš. An approach to model checking AsmL specifications. In *Proc. of 12th International Workshop on Abstract State Machines (ASM'05)*, LNCS, pages 289–304. Springer, 2005.
22. M. Kardoš. *Automated formal verification for UML-based model driven design of embedded systems*. PhD thesis, Slovak University of Technology, 2006.
23. 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.
24. OMG Model Driven Architecture (MDA). `http://www.omg.org/mda`.
25. 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.
26. Rational Software Corporation. *C Reference. Rational Rose RealTime. Version: 2003.06.00, Part no: 800-026108-000*, 2003.
27. Rational Software Corporation. *C++ Reference. Rational Rose RealTime. Version: 2003.06.00, Part no: 800-026109-000*, 2003.
28. Rational Software Corporation. *Java Reference. Rational Rose RealTime. Version: 2003.06.00, Part no: 800-026110-000*, 2003.
29. Rational Rose RealTime tool. Shipped within Rational Rose Technical Developer: `http://www.ibm.com/software/awdtools/developer/technical`.
30. 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.
31. 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.
32. B. Selic. Turning clockwise: using UML in the real-time domain. *Comm. of the ACM*, 42(10):46–54, Oct. 1999.
33. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

34. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.ibm.com/developerworks/rational/library/139.html`, March 1998.
35. SpecExplorer tool (Microsoft). `http://research.microsoft.com/SpecExplorer`.
36. M. von der Beeck. A formal semantics of UML-RT. In *Proc. MoDELS*, volume 4199 of *LNCS*, pages 768–782. Springer, 2006.