Contracts for Multi-instance UML Activities

Vidar Slåtten and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway.
{vidarsl,herrmann}@item.ntnu.no

Abstract. We present a novel way of encapsulating UML activities using interface contracts, which allows to verify functional properties that depend on the synchronization of parallel instances of software components. Encapsulated UML activities can be reused together with their verification results in SPACE, a model-driven engineering method for reactive systems. Such compositional verification significantly improves the scalability of the method. Employing a small example of a load balancing system, we explain the semantics of the contracts using the temporal logic TLA. Thereafter, we propose a more easily comprehensible graphical notation and clarify that the contracts are able to express the variants of multiplicity that we can encounter using UML activities. Finally, we give the results of verifying some properties of the example system using the TLC model checker.

1 Introduction

A key to efficient software engineering is the reuse of existing software components that ideally are independently developed and marketed as commercial off-the-shelf (COTS) products. To enable a seamless combination of the components, one needs meaningful descriptions of the component interfaces that specify properties to be kept by both the components themselves and their environments. Due to the tight interaction with their environment, this requirement holds in particular for reactive systems [6]. To support the reuse of software components for reactive, distributed software systems, we use collaborative building blocks described by multi-partition UML activities, each augmented with an interface contract in the form of a UML state machine [12, 13]. We call these contracts External State Machines (ESMs). The contracts not only enable reuse of problem solutions, but also make for a reuse of verification effort as the user can verify a composed system using only the contracts, which in turn have been verified to be correct abstractions of the underlying solutions. This compositional approach helps to reduce the complexity and state space of the system models significantly [12]. The ESMs also help another problem with reuse: It may not always be straightforward to look at a reusable activity and see what it does and how to compose it correctly with the rest of the system. As the ESM only describes behaviour visible to the outside of the block, it aids both these tasks.

ESMs constitute what Beugnard et al. [2] call synchronization contracts, meaning that they can specify the effect of interleaving operations on a component, not just sequential operations. However, up until now we have been limited to collaborations in which only one instance of each type participates, as the contracts could not support collaborations featuring multiple component instances of the same type. If, for instance, a client request may be routed to one of several servers, we could not express an interface behaviour that permits server S to receive the request only if none of the other servers have received it. In systems that employ load balancing or fault-tolerance mechanisms, however, to specify and guarantee this kind of behaviour is crucial. Thus, compared with the ESMs, we need additional concepts and notation for behavioural interfaces.

Any extension of ESMs should ideally keep a key characteristic of SPACE [14]: The underlying formalism is hidden to the user. According to Rushby [21], this is a key quality of practical development methods. SPACE relies on automatic model checking to verify system models. To mitigate the problem of state-space explosion, we limit our scope to verifying properties dependent only on the control flow of the system designs. While we could very well include data in the model, the model checker would not be able to verify properties dependent on data for realistic systems, as the state space would grow exponentially with every data element we include. Nevertheless, as pointed out in [13, 14], also the model checking of just control flows is of great practical help, for single-instance activities.

The ESMs are basically Finite State Machines (FSMs) with special annotations of their transitions. To model multiple entities in a suitable way, we use Extended Finite State Machines (EFSMs) [3] instead, which allow to refer to the indexes of instances in the form of auxiliary variables. The semantics of these Extended ESMs (EESMs) is formalized using the Temporal Logic of Actions, TLA [15]. To relieve the software engineer from too much formalism, we further present a more compact graphical notation in the form of UML state machines where statements closer to programming languages are used to describe variable operations.

The next section discusses related work on component contracts, particularly work using UML. Our load balancing example system is presented in Sect. 3. In Sect. 4, we formalize the EESM semantics for many-to-many activities in TLA, and present the graphical notation. We give EESMs for the other types of activities, one-to-one and one-to-many, in Sect. 5. Some results, in particular about the effects on model checking, as well as future work is discussed in Sect. 6, where we also conclude.

2 Related Work

There are several other works that define a formal semantics for UML activities [4, 5, 23], but neither of them include contracts for use in hierarchical activities. Eshuis [4] explicitly argues to leave this concept out, as any hierarchical activity can be represented as a flat one. However, this results in a much bigger state space.

As Beugnard et al. [2] point out, we can only expect software components to be reused for mission-critical systems if they come with clear instructions on how to be correctly reused and what guarantees they give under those conditions. UML has the concept of Protocol State Machines [19] to describe the legal communication on a port of a component. Mencl [18] identifies several shortcomings of these, for example that they do not allow to express dependencies between events on a provided and required interface, nor nesting or interleaving method calls. To remedy this, he proposes Port State Machines, where method calls are split into atomic request and response events. These Port State Machines are restricted to pure control flow, as transition guards are not supported. Bauer and Hennicker [1] describe their protocol style as a hybrid of control flow and data state. However, they also cannot express the dependency between provided and required interfaces, and they currently lack verification support for whether two components fit together.

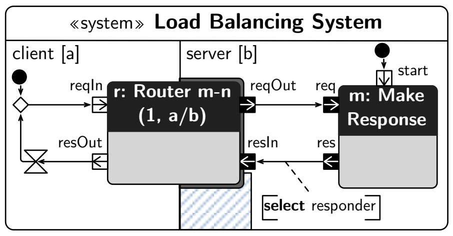
The ESMs have similar properties to Port State Machines in that all interface events are atomic, i.e., an operation is split into a request and response event, to allow for expressing nesting and interleaving of operations. They also essentially combine provided and required interfaces in the same contract, hence allowing to express dependencies between them. The EESMs combine this with data state to allow for compact representations of parametrized components and collaborations.

Sanders et al. [22] present what they call semantic interfaces of service components, both described as finite state machines. These interfaces support both finding complementary components and implementations of an interface, hence also compositional verification. While they provide external interfaces for each local component and then asynchronously couple these to other, possible remote, components, our activity contracts act as internal interfaces that can be synchronously coupled with other local behaviour in the fashion of activity diagrams.

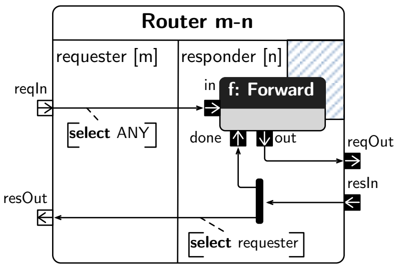
Our approach differs from all the above in that it permits the encapsulation of both local components and distributed collaborations between components, described by single-partition or multi-partition activities respectively. Further, our extended interfaces allow to constrain behaviour based on the state of parallel component instances, giving a powerful abstraction of distributed collaborations.

3 A Load Balancing Client – Server Example

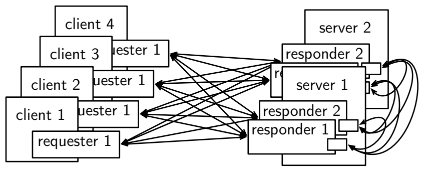
In SPACE, the main units of composition are collaborative building blocks in the form of UML activities that can automatically be transformed to executable code [14]. A system specification consists of a special system activity as the outermost block. This system activity can contain any number of inner activities, referenced by call behaviour actions, as well as glue logic between them. Figure 1(a) shows the system activity for our example, a set of clients that can send



(a) System activity



(b) Router m-n activity



(c) Structural view of system example with four clients and two servers

Fig. 1: System example: A load balancing client – server system

requests via a Router m-n block to a set of servers. Thus, Router m-n is an inner block, its activity depicted in Fig. 1(b). Each activity partition is named in the upper left corner and the bracketed parameter behind the name, a for client and b for server, denotes the number of component instances of this type. While each client only sees one router, each server partition has a/b instances of the router block, as denoted by parameters *(1, a/b)* after its name and the shade around the server side of the block. This is also illustrated in Fig. 1(c), where we see that each client component only has a single requester sub-component, whereas each server has two responders. Note that the structural view is completely redundant and only serves to illustrate the information in the activities and the EESMs introduced below. The diagonally striped area inside the server partition represents other components of the same type, i.e., other servers. This gives a visual indication that the Router m-n block, in addition to collaborating with clients, also collaborates with other server instances. Each server also makes use of an inner activity called Make Response, which turns requests into responses. We have omitted it in Fig. 1(c), as it is completely local.

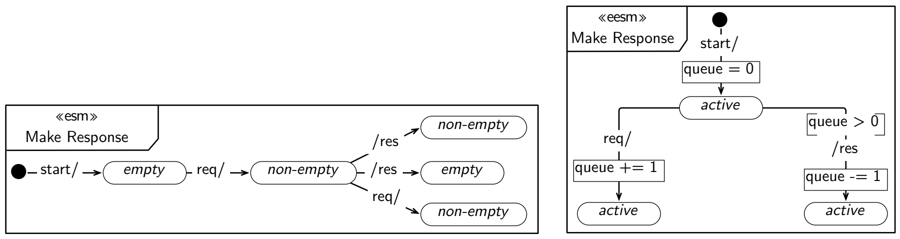
It is the job of the Router m-n block in Fig. 1(b) to balance the load so that a single server does not have to serve requests from all clients at the same time. All requesters can send requests to all responders, as illustrated by the full mesh connectivity in Fig. 1(c). Each responder uses the Forward block to forward requests to other responders, if it is currently busy itself. In the structural view, the component of the Forward block is shown as a small nameless rectangle on each responder that can communicate with every other such component. It is important to note that the Router m-n activity encapsulates asynchronous communication between components, while the synchronous interaction between an outer and an inner activity takes place via pins[1] linking flows of the two activities. For instance, the block Router m-n is linked with the system activity by the pins reqIn, reqOut, resIn and resOut. Here, reqIn is a start pin initiating an activity instance (really, the corresponding requester instance), whereas resOut is a stop pin terminating the instance. The remaining pins with black background are streaming pins for interacting with active instances.

The semantics of UML activities is similar to Petri-nets, where the state is encoded as tokens resting in token places and then moving along the directed edges to perform a state transition [19]. In our approach, all behaviour takes place in run-to-completion steps [9]. That is, all token movements are triggered by either receptions of external events (for instance, tokens resting between partitions) or expiration of local timers, and the tokens move until such an event is needed to make progress again.

Initial nodes start the behaviour of each system-level partition instance. They are fired one by one, but we make the assumption that no token will enter a partition before its initial node has fired. The initial node of the server partition can fire at any time, releasing a token into the start pin of the Make Response block. In the client partition, the token emitted from the initial node will enter the Router m-n block via the reqIn pin. Afterwards, it will be forwarded to a server instance and leave the block via pin reqOut, to enter pin req of Make Response. The Make Response block will eventually emit a token via its res pin, and the server partition will choose one of the Router m-n instances to receive it via its resIn pin, as denoted by the select statement [10]. A select statement takes some data carried by the token and uses it to select either among various instances of a local inner block or of remote partitions. The Router m-n block will eventually emit a token through its resOut pin in one of the client partitions, which will follow the edge to the timer on the client side, where the step will stop. Later, the timer will expire, causing it to emit the token so that the client can perform another request. In this example, the timer is simply an abstraction for whatever the client might be doing between receiving a response and sending a new request. The behaviour of the Router m-n block is described in Sect. 4.

When we compose a system by creating new activities and reusing existing ones, we want to be able to verify properties of it. Given that SPACE models have a formal semantics [11, 12], we can express them as temporal logic specifications and use a model checker to verify properties automatically. To mitigate the problem of the state space growing exponentially with every stateful element, each activity is abstracted by an External State Machine (ESM), which is a description of the possible ordering of events visible on the activity pins. This allows us to do compositional verification of the system specification: We first

[1] They are really activity parameter nodes when seen from the inner activity itself, but are called pins when an activity is reused via a call behaviour action. We use *pins* to denote both, to keep it simple.



(a) ESM (b) EESM

Fig. 2: Contracts for the Make Response block

verify that the activity and its ESM are consistent, then we only use that ESM when verifying properties for a higher-level or system-level block. Note that verifying the consistency of an ESM and its activity cannot be done automatically for all blocks, as some will constrain their control-flow behaviour according to data stored in the activity tokens. In this case, the model allows all possible behaviours, and the potentially false positives reported by a model checker (that is, error traces that cannot occur in the real system) can be inspected manually, reducing the verification task. A select statement is an example where data constrains the destination of a token.

Figure 2(a) shows the ESM of the local building block Make Response. As discussed in the introduction, the ESM notation has the same expressive power as a finite state machine or Mealy machine [17], to be precise. The transition labels correspond to pins, and the slash character separates the transition trigger event, as seen from the perspective of the block, from the effect. Hence, *start/* means that the transition is triggered from outside by a token entering pin start and that no tokens will pass any other pins in the same step. The ESM shows that a response is not output until a request has been sent in, and that this will not happen in the same step. Once the non-empty state is reached, further requests may enter and responses may be emitted from the block. Just looking at the ESM, however, we cannot know exactly how many responses will be sent out, as there is no way of knowing if a /res event has caused a transition to the empty state, or if the ESM is still in the non-empty state. This is because there is no way to track the actual number of buffered requests.[2] When verifying properties of a system, such information may sometimes be necessary. For example, if this block was used in a system that sends three requests, we would like to infer from the ESM that exactly three responses can be emitted back out.

[2] An ESM could of course track the number of buffered requests in explicit states, up to some finite number, but it would quickly grow syntactically very large.

4 Contracts for Multi-instance Activities

To support multi-instance activities, we extend the ESMs to include transition guards, variables, arithmetic and predicate logic. Hence, they are now formally EFSMs [3]. This enables us to specify event constraints that relate to the state of parallel component instances. Moreover, this increases the general expressiveness, so that we are able to better handle the case of the Make Response block.

Figure 2(b) shows the EESM of the Make Response block. We have here added a variable, *queue*, that tracks the number of requests buffered. To constrain the behaviour based on the queue size, as well as update it, this EESM also contains transition guards in square brackets and variable operations in lined boxes.

Figure 1(b) shows the internal activity of the Router m-n block. A request enters through the reqIn pin of the requester partition and is forwarded to a responder partition. The select statement, along with the fact that there are n responder partitions, tells us that a requester expects to have a choice of responders to communicate with, when forwarding the token. When a token crosses a partition border, the corresponding activity step ends, as remote communication is asynchronous. When the token is received by the responder partition, it is passed on to an inner block, Forward. This block may emit the token directly through its out pin to be passed on through the reqOut pin of Router m-n, or it may forward the token to another responder, if this one is busy already serving a request.[3] When a response token is received via the resIn pin, it is duplicated in the fork node and passed both to pin done and the channel for the originating requester partition.

We now describe the EESM of block Router m-n using the language TLA$^+$ of the temporal logic TLA [15], as shown in Fig. 3. The TLA$^+$ module starts by defining the module name on the first line. The EXTENDS keyword states that this module imports the modules Naturals, which defines various arithmetic operators on natural numbers, and MatrixSum, defining operators for summing matrices. The variables of the module are declared using the VARIABLES keyword, where *req* and *res* represent the requester and responder partitions respectively. Constants are declared by the CONSTANTS keyword. They are the parameters of the model. When creating the building block Router m-n, we do not know how it will be reused in an enclosing activity. Another developer may choose to put multiple instances in both, one or none of the enclosing partitions. So, we need constants for the number of requesters and responders per enclosing partition instance, as well as the number of enclosing partition instances on each side. Hence, the global number of requesters is really $no\_req * no\_req\_encl$.

A TLA$^+$ specification describes a labelled transition system. This is given as a set of states, a subset of the states that are initial states, a set of actions (labels) and a transition relation describing whether the system can make a transition from one state to another via a given action. The set of initial states

[3] This behaviour is described by the EESM of the Forward block, which, due to space constraints, we do not show.

```
MODULE $router\_m\_n$
EXTENDS $Naturals,\ MatrixSum$
VARIABLES $req,\ res$
CONSTANTS $no\_req,\ no\_res,\ no\_req\_encl,\ no\_res\_encl$

$Init\ \triangleq$
$\land\ req = [reqIn \mapsto [e \in 1 \,..\, no\_req\_encl,\ i \in 1 \,..\, no\_req \mapsto 0]]$
$\land\ res = [reqOut \mapsto [e \in 1 \,..\, no\_res\_encl,\ i \in 1 \,..\, no\_res \mapsto 0],$
    $resIn \mapsto [e \in 1 \,..\, no\_res\_encl,\ i \in 1 \,..\, no\_res \mapsto 0]]$

$reqIn(e,\ i)\ \triangleq\ req.reqIn[e,\ i] = 0$
$\land\ req' = [req\ \text{EXCEPT}\ !.reqIn[e,\ i] = 1] \land \text{UNCHANGED}\ \langle res \rangle$

$reqOut(e,\ i)\ \triangleq\ res.reqOut[e,\ i] = 0$
$\land\ Sum(req.reqIn,\ no\_req\_encl,\ no\_req) > Sum(res.reqOut,\ no\_res\_encl,\ no\_res)$
$\land\ res' = [res\ \text{EXCEPT}\ !.reqOut[e,\ i] = 1] \land \text{UNCHANGED}\ \langle req \rangle$

$resIn(e,\ i)\ \triangleq\ res.reqOut[e,\ i] > 0 \land res.resIn[e,\ i] = 0$
$\land\ res' = [res\ \text{EXCEPT}\ !.resIn[e,\ i] = 1] \land \text{UNCHANGED}\ \langle req \rangle$

$resOut(e,\ i)\ \triangleq\ req.reqIn[e,\ i] > 0$
$\land\ \exists f \in 1 \,..\, no\_res\_encl,\ k \in 1 \,..\, no\_res :$
    $\land\ res.resIn[f,\ k] > 0$
    $\land\ res' = [res\ \text{EXCEPT}\ !.reqOut[f,\ k] = 0,\ !.resIn[f,\ k] = 0]$
$\land\ req' = [req\ \text{EXCEPT}\ !.reqIn[e,\ i] = 0]$
```

Fig. 3: TLA$^+$ module for the EESM of Router m-n

is described by the *Init* construct. Here, the *req* and *res* variables are each given records for their corresponding pins (except pin resOut, see below), which in turn are functions in two dimensions stating whether a token has passed the pin for each requester or responder instance. That is, a requester or responder instance is identified by an enclosing instance number combined with an inner instance number.

Next in the TLA$^+$ module follow the actions, which formally are predicates on pairs of states. Variables denoting the state before carrying out an action use their common identifiers while those referring to the state afterwards are given a prime. The action *reqIn* states that for a requester$\langle \mathrm{e}, \mathrm{i} \rangle$ identified by enclosing instance e and inner instance i, a token can enter pin reqIn only if the given instance has not yet had a token pass through this pin. The next conjunct of the reqIn action says that the values of the req variable will be the same as now, except that the counter for tokens having passed through pin reqIn will be set to 1 for requester$\langle \mathrm{e}, \mathrm{i} \rangle$. The UNCHANGED keyword states which variables are not changed by an action, as TLA$^+$ requires all next-state variable values to be set explicitly.

The *reqOut* action also represents that a token is only allowed through pin reqOut of responder$\langle \mathrm{e}, \mathrm{i} \rangle$ if it has not already had a token pass through. The second line constrains this further by stating that a token passing event can only happen if the sum of all tokens having passed any reqIn pin is greater than the sum of all tokens having passed any reqOut pin. Hence, this event on responder$\langle \mathrm{e}, \mathrm{i} \rangle$ is constrained by the state of parallel components. Action *resIn*

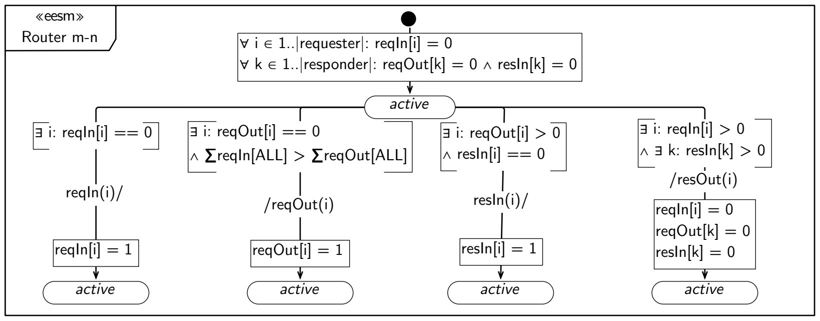


Fig. 4: UML notation for the EESM of Router m-n

states that a token may only enter a responder$\langle \mathrm{e}, \mathrm{i} \rangle$ via pin resIn if the same instance has already emitted a token via pin reqOut, but not already sent a response through pin resIn. The *resOut* action states that a requester$\langle \mathrm{e}, \mathrm{i} \rangle$ can only emit a token through pin resOut if it has received a token through pin reqIn. This is constrained further by requiring there to be a responder$\langle \mathrm{f}, \mathrm{k} \rangle$, that has received a token through its resIn pin. All counters belonging to requester$\langle \mathrm{e}, \mathrm{i} \rangle$ and responder$\langle \mathrm{f}, \mathrm{k} \rangle$ are then set to 0, to reset their state. As the resOut action also performs the reset, there is no TLA$^+$ variable for its corresponding pin.

This behaviour is easier seen looking at the graphical notation in Fig. 4. Here, the style of the transition operations is closer to programming languages like Java. The number of partition instances is denoted |partition name|. We omit the domain of ∃ statements where this is obvious from the context, and the keyword ALL denotes all indexes in a domain. We also omit specifying which partition a pin belongs to if there is only one pin by that name in the activity. The transition from the initial node to the *active* state represents the *Init* construct in TLA$^+$, and the remaining transitions represent the actions.

Since we do not model that a token keeps the index of its requester instance as data while located at a server, we cannot fully automatically verify that the activity and EESM for Router m-n are consistent. The model checker finds counterexamples where a response is simply sent to a requester that has not yet issued a request, instead of to a requester that has. What we can verify automatically, however, is that whenever a token is sent back to a requester, the EESM is in a state where the token would be allowed through the resOut pin of at least one of them.

Once a building block is complete, we can reuse it like we have reused Router m-n in our system example from Fig. 1(a). To verify properties of the system, we generate the TLA$^+$ module in Fig. 5 (see [13]). This module instantiates other modules, namely Router m-n and Make Response. The constants *no_clients* and *no_servers* represent the parameters a and b from the system activity. We express the actions of the system activity as a composition of constraints and operations on the variables of the system activity, and actions of ESMs of the inner activities. Hence, the Init construct not only sets the timer in all client

MODULE *load_sharing_system*

$\textsc{extends}\ \mathit{Naturals},\ \mathit{MatrixSum}$

$\textsc{variables}\ \mathit{r\_req},\ \mathit{r\_res},\ \mathit{m\_state},\ \mathit{m\_queue},\ \mathit{client},\ \mathit{server}$

$\textsc{constants}\ \mathit{no\_clients},\ \mathit{no\_servers}$

$\mathit{no\_res} \triangleq \mathit{no\_clients} \div \mathit{no\_servers}$

$r \triangleq \textsc{instance}\ \mathit{router\_m\_n}\ \textsc{with}\ \mathit{no\_req} \leftarrow 1,\ \mathit{no\_req\_encl} \leftarrow \mathit{no\_clients},$
$\quad \mathit{no\_res} \leftarrow \mathit{no\_res},\ \mathit{no\_res\_encl} \leftarrow \mathit{no\_servers},\ \mathit{req} \leftarrow \mathit{r\_req},\ \mathit{res} \leftarrow \mathit{r\_res}$

$m \triangleq \textsc{instance}\ \mathit{make\_response}\ \textsc{with}\ \mathit{no\_make\_response} \leftarrow 1,\ \mathit{no\_enclosing} \leftarrow \mathit{no\_servers},$
$\quad \mathit{state} \leftarrow \mathit{m\_state},\ \mathit{queue} \leftarrow \mathit{m\_queue}$

$\mathit{Init} \triangleq \mathit{client} = [\mathit{timer} \mapsto [i \in 1 \mathrel{..} \mathit{no\_clients} \mapsto 0],\ \mathit{initial} \mapsto [i \in 1 \mathrel{..} \mathit{no\_clients} \mapsto 1]]$
$\land\ \mathit{server} = [\mathit{initial} \mapsto [i \in 1 \mathrel{..} \mathit{no\_servers} \mapsto 1]] \land r\,!\mathit{Init} \land m\,!\mathit{Init}$

$\mathit{start\_client}(p) \triangleq \mathit{client.initial}[p] = 1 \land \mathit{client}' = [\mathit{client}\ \textsc{except}\ !.\mathit{initial}[p] = 0]$
$\land\ r\,!\mathit{reqIn}(p,\ 1) \land \textsc{unchanged}\ \langle \mathit{server},\ \mathit{m\_state},\ \mathit{m\_queue} \rangle$

$\mathit{start\_server}(p) \triangleq \mathit{server.initial}[p] = 1 \land \mathit{server}' = [\mathit{server}\ \textsc{except}\ !.\mathit{initial}[p] = 0]$
$\land\ m\,!\mathit{start}(p,\ 1) \land \textsc{unchanged}\ \langle \mathit{client},\ \mathit{r\_req},\ \mathit{r\_res} \rangle$

$\mathit{r\_reqOut\_m\_req}(p,\ i) \triangleq r\,!\mathit{reqOut}(p,\ i) \land m\,!\mathit{req}(p,\ 1) \land \textsc{unchanged}\ \langle \mathit{client},\ \mathit{server} \rangle$

$\mathit{m\_res\_r\_resIn}(p,\ i) \triangleq m\,!\mathit{res}(p,\ 1) \land r\,!\mathit{resIn}(p,\ i) \land \textsc{unchanged}\ \langle \mathit{client},\ \mathit{server} \rangle$

$\mathit{r\_resOut\_client\_timer}(p) \triangleq r\,!\mathit{resOut}(p,\ 1) \land \mathit{client.timer}[p] = 0$
$\land\ \mathit{client}' = [\mathit{client}\ \textsc{except}\ !.\mathit{timer}[p] = 1] \land \textsc{unchanged}\ \langle \mathit{server},\ \mathit{m\_state},\ \mathit{m\_queue} \rangle$

$\mathit{client\_timer\_r\_reqIn}(p) \triangleq \mathit{client.timer}[p] = 1 \land \mathit{client}' = [\mathit{client}\ \textsc{except}\ !.\mathit{timer}[p] = 0]$
$\land\ r\,!\mathit{reqIn}(p,\ 1) \land \textsc{unchanged}\ \langle \mathit{server},\ \mathit{m\_state},\ \mathit{m\_queue} \rangle$

$\mathit{Next} \triangleq$
$\lor\ \exists\, p \in 1 \mathrel{..} \mathit{no\_clients} : \mathit{start\_client}(p)$
$\lor\ \exists\, p \in 1 \mathrel{..} \mathit{no\_servers} : \mathit{start\_server}(p)$
$\lor\ \exists\, p \in 1 \mathrel{..} \mathit{no\_servers},\ i \in 1 \mathrel{..} \mathit{no\_res} : \mathit{r\_reqOut\_m\_req}(p,\ i)$
$\lor\ \exists\, p \in 1 \mathrel{..} \mathit{no\_servers},\ i \in 1 \mathrel{..} \mathit{no\_res} : \mathit{m\_res\_r\_resIn}(p,\ i)$
$\lor\ \exists\, p \in 1 \mathrel{..} \mathit{no\_clients} : \mathit{r\_resOut\_client\_timer}(p)$
$\lor\ \exists\, p \in 1 \mathrel{..} \mathit{no\_clients} : \mathit{client\_timer\_r\_reqIn}(p)$

$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{r\_req},\ \mathit{r\_res},\ \mathit{m\_state},\ \mathit{m\_queue},\ \mathit{client},\ \mathit{server} \rangle}$

$P1 \triangleq \Box(\forall\, p \in 1 \mathrel{..} \mathit{no\_servers} : \mathit{m\_queue}[p,\ 1] \leq \mathit{no\_res})$

$P2 \triangleq \Box(\mathit{Sum}(\mathit{r\_res.reqOut},\ \mathit{no\_servers},\ \mathit{no\_res}) \leq \mathit{Sum}(\mathit{r\_req.reqIn},\ \mathit{no\_clients},\ 1))$

$P3 \triangleq \Box(\forall\, p \in 1 \mathrel{..} \mathit{no\_servers},\ i \in 1 \mathrel{..} \mathit{no\_res} :$
$\quad (\mathit{server.initial}[p] = 0 \land \textsc{enabled}\ r\,!\mathit{reqOut}(p,\ i)) \Rightarrow \textsc{enabled}\ m\,!\mathit{req}(p,\ 1))$

$P4 \triangleq \Box(\forall\, p \in 1 \mathrel{..} \mathit{no\_servers} : \textsc{enabled}\ m\,!\mathit{res}(p,\ 1) \Rightarrow$
$\quad \exists\, i \in 1 \mathrel{..} \mathit{no\_res} : \textsc{enabled}\ r\,!\mathit{resIn}(p,\ i))$

$P5 \triangleq \Box(\forall\, p \in 1 \mathrel{..} \mathit{no\_clients} : \textsc{enabled}\ r\,!\mathit{resOut}(p,\ 1) \Rightarrow \mathit{client.timer}[p] = 0)$

Fig. 5: TLA$^+$ module for the system activity

instances to 0 and all initial nodes to 1, but also calls the Init construct of Router m-n and Make Response as shown by $r!Init$ and $m!Init$. Note also that since this is a system activity, we do not need to add an extra dimension for enclosing partition instances when identifying activity elements like the timer, as it cannot be reused in other activities. For a description of each action, we refer back to Sect. 3.

The whole system specification is written as a single formula $Spec \triangleq Init \land \Box[Next]_{\langle vars \rangle}$. This formula states that the transition system has initial states(s) as specified by *Init* and that every change to the variables listed in *vars* is done via one of the actions listed in the next-state relation, *Next*. The box-shaped symbol ($\Box$) in front of [*Next*] is the temporal logic operator *always*. It means that what follows must be true for every reachable state of the system model.

The small example system of this paper is chosen to allow us to show the formal semantics of the EESMs in TLA$^+$ and clarify that they are unambiguous, yet expressive enough for our needs. Therefore, the properties that we can verify for this system might seem rather trivial, but for more complex systems, variations of these properties may be very hard to verify without a formal model. The properties we want to verify are written formally below the horizontal bar in Fig. 5. All the properties can be verified by model checking. See Sect. 6 for further discussion of the results.

P1 The number of requests queued in any Make Response block is at most equal to the number of responders per server, i.e., the inner queue is finite.

P2 There are at most as many ongoing requests on the server side as there are on the client side.

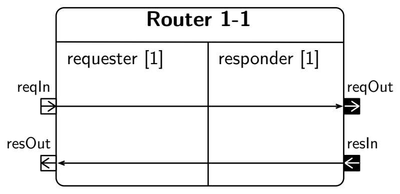
P3 Whenever a server is started and a responder instance of that server is ready to emit a token through the reqOut pin, the Make Response instance of that server is ready to accept a token through its req pin.

P4 Whenever a token can be emitted from the Make Response block of a server, at least one of the responder instances on that server is able to accept it.

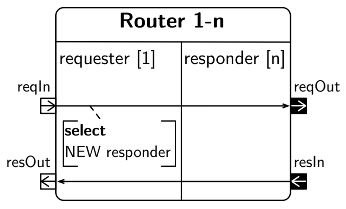
P5 Whenever a token can be emitted via the resOut pin of a requester instance, the corresponding timer is empty, hence ready to receive a token.

5 Other Types of Multiplicity

Our formalism for expressing contracts of multi-instance activities also works for one-to-one building blocks without any internal select statements, like Router 1-1 shown in Fig. 6(a). This is a special case, where each requester instance is statically mapped to a responder instance and vice versa. As each binary collaboration cannot have any constraints on its behaviour in terms of the state of parallel instances, we can simplify the EESM as shown in Fig. 7 without loss of information. This is, in fact, the same notation that we have been using already for ESMs of activities with one instance of each type [14], only augmented with an index i. The difference is that the formal semantics now supports multiple instances globally, instead of requiring such a block to be used in a system with only one instance of each enclosing partition as well.



(a) Router 1-1



(b) Router 1-n

Fig. 6: The two other variants of the router activity, with respect to select statements

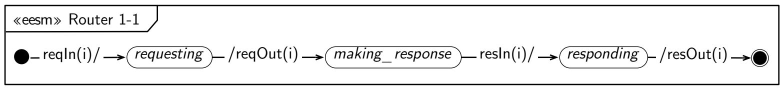


Fig. 7: UML notation for the EESM of Router 1-1

Finally, we present a one-to-many variation of the router block, Router 1-n, where a requester is statically mapped to a set of responders, as shown in Fig. 6(b).[4] This could be used in a setting where each server from Fig. 1(a) has one responder instance per client, so that each client has a choice of any server when issuing a request. When the response is to be routed back, the corresponding requester is already given, due to the static mapping.

The EESM of Router 1-n is shown in Fig. 8. Due to the mapping between requester and responder instances, the notation is a bit more complex than for the other variants. For example, the /reqOut(i) transition states that a token may only leave the reqOut pin through instance i if this has not happened already. The rest of the guard constrains this further by stating that a token passing can only occur if the corresponding requester instance has gotten more tokens through its reqIn pin than the sum of tokens having already passed through pin reqOut in all the responders mapping to that requester. The expression $reqIn[responder[i]]$ means the value of the reqIn variable for the requester who can be found by mapping $responder[i]$ to its corresponding requester. Hence, $\Sigma\ reqOut[requester[responder[i]][k]]$ means the sum of reqOut values for the k different responders found by mapping responder[i] to its requester and then mapping that requester to the set of corresponding responders.

Note that it is the EESM that holds the information on whether there is a constrained static mapping or not. In contrast, the EESM of Router m-n, given in Fig. 4, has no references to any particular parallel instance or set of instances, only to the current instance and the keyword ALL.

[4] In addition, there could naturally be a mirror version of the Router 1-n activity, a Router n-1, but this is also a one-to-many activity.

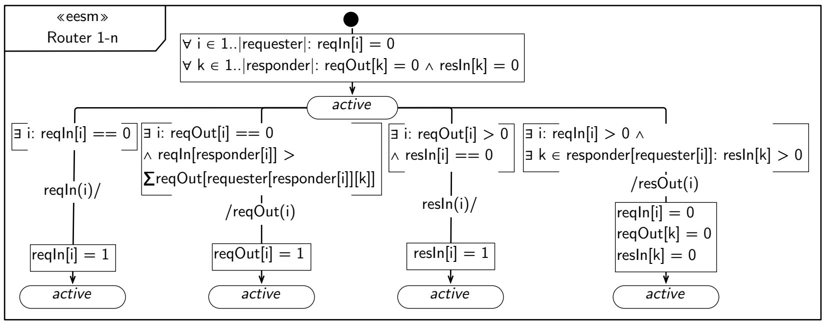

«eesm»
Router 1-n
∀ i ∈ 1..|requester|: reqIn[i] = 0
∀ k ∈ 1..|responder|: reqOut[k] = 0 ∧ resIn[k] = 0
active
∃ i: reqIn[i] == 0
reqIn(i)/
reqIn[i] = 1
∃ i: reqOut[i] == 0 ∧ reqIn[responder[i]] > ΣreqOut[requester[responder[i]][k]]
/reqOut(i)
reqOut[i] = 1
∃ i: reqOut[i] > 0 ∧ resIn[i] == 0
resIn(i)/
resIn[i] = 1
∃ i: reqIn[i] > 0 ∧ ∃ k ∈ responder[requester[i]]: resIn[k] > 0
/resOut(i)
reqIn[i] = 0
reqOut[k] = 0
resIn[k] = 0
active

Fig. 8: UML notation for the ESM of Router 1-n

Table 1: Number of states found and time required to verify properties P1–P5

# of servers → # of clients ↓	1	2	3
1	7 states, < 1 sec		
2	37 states, 1 sec	70 states, 1 sec	
3	241 states, 2 sec		707 states, 3 sec
4	1713 states, 4 sec	3410 states, 5 sec	
5	12617 states, 9 sec		
6	94513 states, 48 sec	188962 states, 99 sec	283411 states, 155 sec
7	715001 states, 651 sec		

6 Concluding Remarks

All the properties from Fig. 5 have been verified by the TLC model checker [24], for the parameter values shown in Table 1.[5] Model checking only verifies properties for a model with some exact parameters. It does not say whether those properties will still hold for different parameters. However, if the model changes behaviour with respect to a property for some specific parameter values, it is often when a parameter is changed from 1 to >1, or it is likely due to highly intentional design decisions. Hence, the fundamental problem remains, but it is not always that great in practice.

Given that model checking is automatic, one could say that time is not an issue, as we can just leave a computer at it and check for up to, for example, a thousand instances of each partition. However, as Table 1 shows, the time needed grows exponentially as we increase the number of client instances. The linear increase from server instances comes from the fact that more servers reduce the number of responders per server.

[5] We are aware that the TLA$^+$ specification for the given example can be optimized by only storing the aggregate number of tokens having passed through a pin on any of the responders in a server. However, this optimization would not work if the EESM required two tokens to pass pin reqOut before a token is allowed though pin resIn.

There is a high level of parallelism in our system example. This is also the case for other systems using EESMs that we have verified. Hence, we expect partial order reduction [7] to alleviate the state-space blowup from increasing the number of instances. We therefore plan to also formalize our semantics in Promela, so we can use the Spin [8] model checker, which implements partial order reduction. The formalisms are compatible, as there is already work for transforming another TLA derivative, cTLA, into Promela automatically [20]. For relatively simple blocks, where the contract must be verified for any number of instances, the TLA formalism allows for writing manual refinement proofs as well [16].

We already have a tool for generating TLA$^+$ from SPACE models [13]. This tool greatly reduces the time required to specify systems, and it automatically generates many types of general properties to ensure the well-formedness of SPACE models. We will extend the tool to work with EESMs, outputting Promela specifications as well. To hide the formalism when specifying application-specific properties, there is work in progress to express them in UML.

To verify properties like "Every request is eventually responded to", would require adding data to identify each request and adding liveness constraints to the model. Being based on TLA, the formalism can accommodate this quite easily in the form of weak or strong fairness assumptions. The limiting factor is still time needed for model checking.

Having formalized extended ESMs, we are eager to use them in the setting of fault-tolerant systems, where multiple instances of the same type often collaborate to mask failures, and conditions such as a majority of the instances being reachable are often essential to precisely describe the behaviour of a block.

To conclude, contracts encapsulate software components and facilitate both reuse and compositional verification. The SPACE method uses collaborations detailed by UML activities as the unit of reuse. We introduce EESMs, which allow to describe the global behaviour of multi-instance activities, abstracting away internal state, while still having the expressive power to detail when an external event can take place. An example from the load balancing Router m-n block is that a request will only arrive at a server that has free capacity in the form of free responder instances, and only if the number of requests received from all clients is greater than the number of requests forwarded to any server. While the EESMs have a formal semantics in TLA, we give graphical UML state machines as specification tools, so that software engineers themselves need not be experts in temporal logic.

References

1. Bauer, S., Hennicker, R.: Views on Behaviour Protocols and Their Semantic Foundation. In: Algebra and Coalgebra in Computer Science, LNCS, vol. 5728, chap. 25, pp. 367–382. Springer Berlin / Heidelberg (2009)
2. Beugnard, A., Jezequel, J.M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. Computer 32, 38–45 (1999)

3. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Proc. 30th Int. Design Automation Conf. pp. 86–91. DAC '93, ACM (1993)
4. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
5. Guelfi, N., Mammar, A.: A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In: Proc. 12th Asia-Pacific SE Conf. pp. 283–290 (2005)
6. Harel, D., Pnueli, A.: On the development of reactive systems. In: Logics and models of concurrent systems, pp. 477–498. Springer New York, Inc. (1985)
7. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proc. 7th IFIP WG6.1 Int. Conf. on Formal Description Techniques. pp. 197–211 (1995)
8. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2003)
9. Kraemer, F., Herrmann, P.: Reactive Semantics for Distributed UML Activities. In: Formal Techniques for Distributed Systems, LNCS, vol. 6117, pp. 17–31 (2010)
10. Kraemer, F.A., Bræk, R., Herrmann, P.: Synthesizing components with sessions from collaboration-oriented service specifications. In: Proc. 13th Int. SDL Forum Conf. on Design for Dependable Systems. pp. 166–185. SDL'07 (2007)
11. Kraemer, F.A., Herrmann, P.: Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In: Proc. Networking and Electronic Conf. (2007)
12. Kraemer, F.A., Herrmann, P.: Automated Encapsulation of UML Activities for Incremental Development and Verification. In: Proc. 12th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS). pp. 571–585. LNCS (2009)
13. Kraemer, F.A., Slåtten, V., Herrmann, P.: Engineering Support for UML Activities by Automated Model-Checking — An Example. In: Proc. 4th Int. Workshop on Rapid Integration of Software Engineering Techniques (RISE) (2007)
14. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software 82(12), 2068–2080 (2009)
15. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
16. Lamport, L.: Refinement in state-based formalisms. Tech. rep., Digital Equipment Corporation, Systems Research Center, Palo Alto, California (1996)
17. Mealy, G.H.: A Method to Synthesizing Sequential Circuits. Bell Systems Technical Journal 34(5), 1045–1079 (1955)
18. Mencl, V.: Specifying Component Behavior with Port State Machines. Electronic Notes in Theoretical Computer Science 101, 129–153 (2004)
19. OMG: Unified Modeling Language: Superstructure, Version 2.3 (2010)
20. Rothmaier, G., Poh1, A., Krumm, H.: Analyzing Network Management Effects with Spin and cTLA. In: Security and Protection in Information Processing Systems, IFIP, vol. 147, chap. 5, pp. 65–81. Springer Boston (2004)
21. Rushby, J.: Disappearing formal methods. High Assurance Systems Engineering, Fifth IEEE International Symposim on pp. 95–96 (2000)
22. Sanders, R.T., Bræk, R., von Bochmann, G., Amyot, D.: Service Discovery and Component Reuse with Semantic Interfaces. In: SDL 2005: Model Driven Systems Design, LNCS, vol. 3530, chap. 6, pp. 1244–1247. Springer (2005)
23. Storrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electronic Notes in Theoretical Computer Science 127(4), 35–52 (2005)
24. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Proc. 10th IFIP WG 10.5 Adv. Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME'99). LNCS, vol. 1703, pp. 54–66 (1999)