Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed Systems

A Method Exemplified by an Industrial Case Study

Vidar Slåtten Frank Alexander Kraemer Peter Herrmann

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

Abstract

The validation and verification of reliable systems is a difficult and complex task, mainly for two reasons: First, it is difficult to precisely state which formal properties a system needs to fulfil to be of high quality. Second, it is complex to automatically verify such properties, due to the size of the analysis state space which grows exponentially with the number of components. We tackle these problems by a tool-supported method which embeds application functionality in building blocks that use UML activities to describe their internal behaviour. To describe their externally visible behaviour, we use a combination of complementary interface contracts, so-called ESMs and EESMs. In this paper, we present an extension of the interface contracts, External Reliability Contracts (ERCs), that capture failure behaviour. This separation of different behavioural aspects in separate descriptions facilitates a two-step analysis, in which the first step is completely automated and the second step is facilitated by an automatic translation of the models to the input syntax of the model checker TLC. Further, the cascade of contracts is used to separate the work of domain and reliability experts. The concepts are proposed with the background of a real industry case, and we demonstrate how the use of interface contracts leads to significantly smaller state spaces in the analysis.

Categories and Subject Descriptors C.0 [*General*]: Systems specification methodology; C.2.4 [*Computer-Communication Networks*]: Distributed Systems—Distributed applications; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Computer-aided software engineering (CASE); D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods, Model checking, Reliability; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Design, Reliability, Verification

Keywords Model-driven engineering, reliable systems, fault tolerance, component contracts, compositional verification, model checking

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

1. Introduction

Since nearly half a century ago, theoretical computer scientists have developed a plethora of techniques to model and to verify software in a formal way. In spite of several outstanding results, however, formal methods are still not used that much in practise. A likely reason is the complexity of many methods which tend not only to be laborious but also require a considerable amount of expertise. Hence, to make the application of formal techniques more popular in software development, they must be much simpler and faster to use. An approach to achieve this is through model-driven engineering, for instance on the basis of UML or SDL. These languages can be effectively used to describe software in such a way that implementations can be automatically generated from them. Further, formal methods may be used to analyze them, as they often give an appropriate level of abstraction. In this way, models can be used as front-ends for formal tools, which leads to what Rushby [24] calls "disappearing formal methods."

Our method SPACE and its tool Arctis [14, 18] are designed with this strategy in mind and optimized for the development of reactive, distributed applications. System behaviour is modelled by UML activities that due to their token semantics similar to Petri-nets [16, 22] can be easily understood. The activities have been provided with a reactive formal semantics [16] such that both automatic code generation [14, 18] and formal analysis [18] are possible. Moreover, this modelling technique is scalable since activities can be composed using UML call behavior actions, which we refer to as *blocks*. A block may both embed an activity and be a part of another one. The interaction between the two activities is modelled by UML pins through which tokens flow when transferring from one activity to the other. The behaviour at this interface is modelled by contracts in the form of so-called External State Machines (ESMs, [15]) and an extended version of them (EESMs, [26]). This allows for storing blocks in libraries and re-using them in different software models (see [15]).

Arctis facilitates the formal analysis of system models for important, generally desirable software properties by using a model checker [18]. Since the model checker does its analysis by an exhaustive search of the reachable system states, it works fully automatically. Further, error traces can be animated on the UML activities such that the Arctis users do not need a deeper understanding of the formalism laying behind the analysis.

A disadvantage of model checking is that models of realistic systems often comprise too many system states to be checked in an acceptable amount of time. This is due to the combinatorial blow-up in the number of states when combining component behaviours, known as the state explosion problem. In Arctis, we mitigate this problem by compositional verification (see [18]). In particular, we

use the interface contracts to reduce the system verification to a number of local verification runs each concerning only a single activity as well as a number of (E)ESMs. Thus, the number of states to be model checked grows linearly with the number of the activities in the system model instead of increasing exponentially.

However, while this strategy works to ensure that each building block is well-formed from a local point of view, for some systems we also need to check application-specific system properties. This holds especially for properties with respect to reliability, i.e., to check how a system reacts in the presence of communication and process failures. The need to consider the system with a larger scope and the presence of failures increases the complexity (and hence, the state space) drastically, simply because of the higher degree of concurrency. Furthermore, mechanisms to improve reliability often employ multiple instances of a given type for redundacy (see, for instance [19]). Here, one has to take also data that distinguishes the individual instances into account, which further escalates the state space to be checked.

This paper is devoted to demonstrating how compositional verification can mitigate the state explosion problem for reliable systems. To achieve that, we validate the concept of EESMs [26] by showing how they are applied in an industrial case study where we were contracted to add a fault-tolerant best-effort mutual exclusion protocol to an existing system of intelligent clothes lockers for hospitals (see Sect. 2). In Sect. 3, we describe our development method for reliable systems, extended from earlier work considering just message loss [27], and the specifications created for the case study. In addition, we show how the activities and contracts are transformed into TLA$^+$ [20], the input language of the TLC model checker [29]. In contrast to proofs of system functionality, not all theorems specifying the reliability properties to be verified can be automatically generated at the moment. Instead, we demonstrate the flexibility of the framework by showing how the best-effort mutual exclusion property can be expressed by an expert (see Sect. 3.2). Furthermore, in Sect. 4, we introduce a novel extension we call External Reliability Contracts (ERCs) as a way to handle the effects of process crashes and message loss in a compositional manner, while still being able to verify larger system sizes under failure-free semantics. We show that our variant of compositional verification gives significant savings over monolithic verification and discuss our findings, in Sect. 5 We survey some related work in Sect. 6 and conclude in Sect. 7.

2. Texi Case Study: Lockers that Read RFID

Texi AS is a company that delivers RFID-based logistics systems to organize work wear, typically for hospitals. All clothes have a small RFID tag sewn into the fabric, which is used to track the clothes during the entire usage cycle. Our case study focuses on the lockers in the hospital that store the clothes and make them available to the hospital staff. The lockers are equipped with antennas that can read the RFID tags, so that they know which clothes are stored in them.

When employees want new clothes, they swipe their employee card through a card reader. The locker then checks if the employee is allowed access to the locker. If the employee has access, the door is unlocked. After the clothes are removed and the door is closed again, the reading process is started to see which clothes have been removed by the employee.

A typical installation in a hospital has many lockers that stand next to each other, and the reading process of the antennas takes several seconds. For that reason it is likely that several employees access closely located lockers simultaneously. This can lead to wrong reading results, since the antennas may interfere once two or more lockers read at the same time.

For this reason, we introduced a solution that delays the reading within lockers so that only one locker may read at a time. Obviously, this can be achieved by introducing a central controller that takes the role of coordinator among all lockers that are in danger of interference. A locker then has to obtain permission before it may activate its antennas. Such a solution, however, introduces several single points of failure. If the central controller goes down, or if a locker does not release its read permission, all further reading requests from other lockers will not be answered and employees will not be able to get their work wear. For that reason, we had to develop a more robust solution, where a locker can carry on alone, if other parts of the system fail.

2.1 System Requirements

According to Texi, the requirements for the improved system are the following: If possible, only one locker shall read its contents at a time. However, availability should still have priority over this mutual exclusion property, since the possible inconsistencies due to concurrent reading can be manually corrected, but a blocked locker would hinder hospital work. Hence, we cannot use a mutual exclusion algorithm that blocks if a locker is unable to contact the central controller. Instead we create a protocol that attempts to provide mutual exclusion of locker reads, but does not necessarily provide it in the presence of process crashes and message loss.

While it is easy to express a mutual exclusion property, *"no two lockers should have permission to read their contents at the same time,"* it is not straightforward to express the kind of best-effort mutual exclusion property (in the following called *BEME* property) that our customer requests. For our protocol, the BEME property can be expressed as a mutual exclusion property that is conditional on the absence of message loss and process crashes. We note that message loss is indistinguishable from a long delay. Thus, such events can be explicitly modelled in the form of timeout events. The same goes for process crashes, which we can model as a transition to a state where no further events can take place in the crashed component of the system, except a restart event. With this in place, we might simply state the BEME property to be *"as long as no timeout or crash has occurred, the mutual exclusion property must hold."* However, this does not say anything about whether the system will recover from a faulty state and go back to providing the mutual exclusion property once a sufficiently long time has passed without further crashes or timeouts taking place. To include this, we can express the BEME property as *"if there is a time after which no further timeouts or crashes occur, then there will eventually be a time where the mutual exclusion property holds forever."* In addition to the BEME property, the customer naturally wants the system to be free from any deadlock scenarios and that all read requests are eventually granted.

Next, we will introduce our method and the specification of the case study before we revisit the BEME property in Sect. 3.2.

3. The Method for Reliable Systems

As explained above, the main specification element of the SPACE method are building blocks, expressed by UML activities and encapsulated by formal contracts. Within the method, we distinguish three levels of descriptions for the external contracts of a building block that have different significance with respect to the overall development workflow:

- External State Machines (ESMs, [15]) are UML state machines that describe the order in which parameter pins on the building blocks may be used. ESMs do not consider any data or sessions (multiple instances of a block), which in many cases is sufficient to explain how blocks are to be assembled correctly to more comprehensive applications. The complexity of the analysis is also limited, so that checking whether the blocks are correctly

integrated can be performed in the background of the editing process, without interrupting the user.

- Extended ESMs (EESM, [26]) are an extension of ESMs that adds actions and guards on data variables. This is especially useful for mechanisms that increase the reliability of systems, since they often require multiple instances of the same block, i.e., sessions. Since session IDs are data, EESMs may capture relations of several sessions, or simply count how often a certain action has been executed. Thus, EESMs are more expressive than ESMs, but this comes at the cost that the verification with EESMs is more complex. This analysis is therefore carried out in an extra step with temporal logic as the basis (see Sect. 3.2).
- External Reliability Contracts (ERCs, introduced in this paper) add yet another layer to the contracts. They amend (E)ESMs by describing behaviour that results from communication and process failures. In principle, this behaviour could be directly expressed within the (E)ESMs. However, we have observed that this behaviour is often orthogonal to the original, purely functional behaviour described by (E)ESMs. It can therefore be expressed separately, much like an aspect in aspect-oriented programming. This also has the benefit that systems may be analyzed both with and without failures, as discussed later.

To make our method practical, we also take into account the level of expertise that is needed to fulfil a certain task. Therefore, we identify two groups of engineers:

- *Domain experts*, who are familiar with a certain application domain and relevant technologies, such as for example RFID and embedded systems. Domain experts have programming skills and may also model on the level of UML activities, but do not need the ability to formulate and verify temporal theorems, for instance.
- *Reliability experts*, who are familiar with reliability problems and possible solutions, and who are also familiar with the necessary formal methods to assure system quality with respect to reliability. To optimally utilize their expertise, we assume that reliability experts are hired on a case-by-case basis. Therefore the method should be optimized to keep their overhead regarding non-reliability related questions low, as we will discuss in the following.

Within a verification project [2], in which we apply our tool to industrial cases such as the one presented, we see that this categorization is quite to the point. With respect to these roles, we expect our method to be used in the following way, as illustrated in Fig. 1:

1. Domain experts create the main part of the application by composing building blocks. We have measured that up to 71% of blocks may be reused from libraries (see [15]). This may already include some building blocks provided by reliability experts, if their necessity is obvious or if the domain expert already knows these blocks from previous projects. For the domain expert it is often sufficient to look at the (simple) ESMs to integrate building blocks correctly into a system.
2. The basic analysis (A_1) is executed in the background, so that the resulting specification S is at least well-formed and describes consistent compositions of the building blocks, as far as the ESMs are concerned. This eases the job of the reliability expert, since many inconsistencies and ambiguities are already removed.
3. The consistent model S is handed over to a reliability expert, who performs an in-depth analysis A_2 using the model checker TLC [29]. TLA$^+$ [20], the input language of TLC, is generated automatically from the UML model, now also including the variables, actions and guards contributed by EESMs. Some systems, such as the one presented in our case study, require also application-specific properties for which the reliability expert formulates the corresponding theorems, as explained in Sect. 3.2.
4. To verify properties also with realistic assumptions including faulty channels and crashing processes, the ERCs of the building blocks are taken into account, or, where necessary, introduced.

   The results of A_2 may require that additional fault-tolerance mechanisms are introduced or given functionality is changed. Depending on the extent of the changes, these are either done by the reliability expert alone, or in cooperation with the domain expert. These decisions may also require feedback from the customer, when consequences of failures and remedies have to be taken into account. In our case study, for instance, Texi had to make the trade-off between data consistency and locker availability.
5. From a consistent specification, executable code can be generated by a model transformation from the UML activities to state machines and a subsequent code generation step. This step is completely automated, and we verified that the code generation preserves system constraints [14]. It is hence guaranteed that this code also fulfils the properties of the original specification.

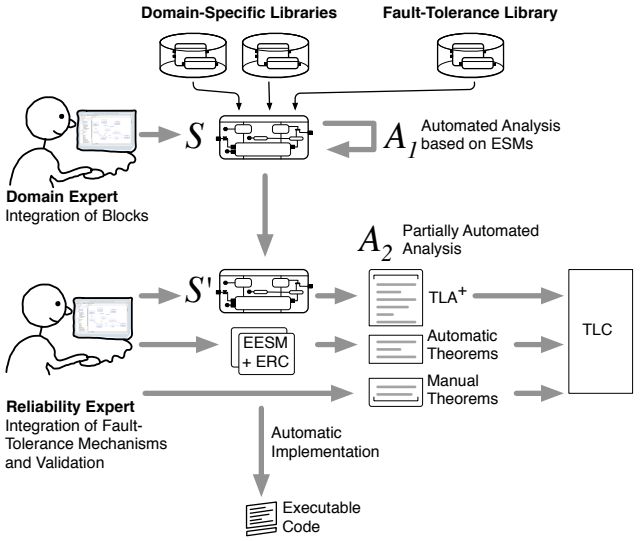

Figure 1. The development method

Needless to say, this method is an idealized workflow that serves as an orientation rather than an inflexible corset. In practise, the separation between the experts may be less distinct than described. In addition, since building blocks may be checked back into the library, complete solutions that already take reliability concerns into account may be directly applied by domain experts. In [19], for instance, we developed a robust leader election protocol. Although the protocol is not trivial, the resulting building block is so easy to handle that it can be integrated by a domain expert without any trouble.

3.1 System Design in SPACE

In the following, we focus on the parts of the system that deal with mutual exclusion between lockers, and do not further detail

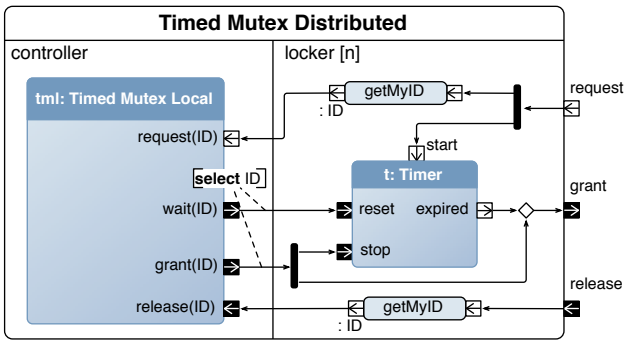

Figure 2. UML activity showing the mutual exclusion protocol

other functionality. We assume that steps 1 and 2 have already been carried out, and focus now on the work of the reliability expert.

Figure 2 shows the UML activity for the Timed Mutex Distributed building block that we created to implement the mutual exclusion protocol. Partitions represent separate components of the system that are physically distributed. They are named in the upper left corner, in this case *controller* and *locker*. The fact that there can be more than one locker is represented by *[n]* after the name of the locker partition. To distinguish the different lockers, each of them has a specific ID. In the implementation, we have chosen the IP address of each locker as its ID. The single controller partition contains a building block *Timed Mutex Local*, which implements the locally concentrated part of the protocol.

UML activities have a semantics based on token flows. For the building blocks, we use a *reactive* variant of them [16], in which tokens flow in run-to-completion steps (so-called *activity steps*), each of which are triggered by an observable event, either the expiration of a timer or the reception of a signal.

Each locker partition of the building block can receive a request for the read permission via the starting pin named *request*, through which a token enters the activity, travels along the edge to the fork node and is duplicated. One duplicate enters the Timer block via pin *start*. The other passes through operation *getMyID*, which retrieves the ID of the locker, and comes to rest at the partition border between the locker and the controller partition. All tokens rest between partitions, as message passing is asynchronous, meaning that the sending and receiving of a message are two different events.

Two activity steps are now enabled: One step is triggered by the token on the topmost edge arriving at its destination and entering the block Timed Mutex Local via pin *request(ID)*. Another possibility is for the timer to emit a token. To see why this can happen, we must consider the external contract of the Timer block.

The ESM of block *Timer* is shown in Fig. 3. It shows that once a token has passed via the *start* pin, the block may spontaneously emit a token via its *expired* pin and will also accept tokens via the pins *stop* and *reset*. Therefore, a possible next event is that the timer expires, releasing a token through the *expired* pin, via the merge node and on through the *grant* pin of the locker partition. Hence, this timer ensures that no locker is blocked from reading its contents by the controller crashing or a communications failure.

Figure 4 shows the EESM of the Timed Mutex Local block, and thus also the behaviour we can expect from it. EESMs are different from ESMs in that their state does not just consist of the control state like *active*, but also the values assigned to any variables they have (i.e., they are *extended* finite state machines [6]). Due to the extra variables, EESMs are initialized implicitly, as shown by the transition from the initial state, at the very left. That is, the initial transition is executed together with the startup of the component (modelled by a top-level partition) the building block is part of. In

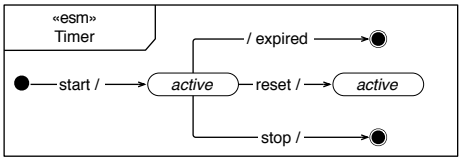

Figure 3. ESM of the Timer block

the case of Timed Mutex Local, the EESM tracks two sets, the IDs that have been sent through pin *request* and the same for pin *grant*. An EESM also takes into account that a block can be instantiated as *multi-session*, which means that several instances of a block execute concurrently. Therefore, every variable of the EESM is an array in which each block instance has its own index, *i*. Note that in Fig. 2, only one instance of the Timed Mutex Local block is instantiated, so *i* is always 1.

The EESM only allows one request from each ID for each block instance at a time. To express this, the transition labelled *request(i, ID)* has a guard stating that for a *request(i, ID)* event to happen, that ID must not already be in the set *requests[i]*. Further, the transition has an operation, written in a lined rectangle, that specifies that the new ID is added to the set represented by *requests[i]*. A *wait(i, ID)* transition can only take place if the block has already received a request from that ID, but not yet sent out a grant. A *grant(i, ID)* transition has the exact same guard as *wait(i, ID)*, but has an additional operation to update the *grants* variable. The *release(i, ID)* transition is only enabled when ID has been granted. This transition resets the contract with respect to that ID by removing it from both the requests and grants set, allowing new requests with that ID.

Looking back at Fig. 2, we see that a token released from the *wait(ID)* pin of block *tml*, denoted *tml:wait(ID)*, will need to rest at the partition border before being received by the locker it is destined for. As there are several lockers, a select statement [17] is used to only send the message to the locker with the address given in the ID parameter of the token. Upon arrival at the locker, the token originating from *tml:wait(ID)* will enter the *reset* pin of the Timer block, hence delaying the expiration of the timer. If the timer has already expired when this happens, the semantics of the contract is that the token is discarded as it attempts to enter the block via a pin that the contract, in its current state, does not allow tokens to pass. During our analysis, such a scenario raises a warning so that the developer can make sure it is intentional.[1]

The events following a token being released via *tml:grant(ID)* are similar to the above, only here, the token is duplicated upon arrival at the locker to both stop the timer and pass through the *grant* pin of the Timed Mutex Distributed block itself. The EESM of Timed Mutex Distributed, shown in Fig. 5,[2] does not permit more than one grant without a release and request in-between, for that locker instance. If a grant has already happened, the token will be discarded when trying to leave the block via the grant pin, just like a token trying to enter a block at the wrong time. The EESM also tells us that a token can enter the Timed Mutex Distributed block via pin *release* if that locker instance has received

[1] Note that this filtering effect only applies when the contract can decide if a transition is enabled based solely on local information. An EESM transition with a guard that refers to a remote variable (e.g., another component instance being in a specific state) cannot filter out tokens, hence any violations are real errors.

[2] When an EESM does not use any data apart from session IDs and transitions do not reference other sessions, we present it in a simpler way that looks like an ESM except for the additional index *(i)* on every transition label [26].

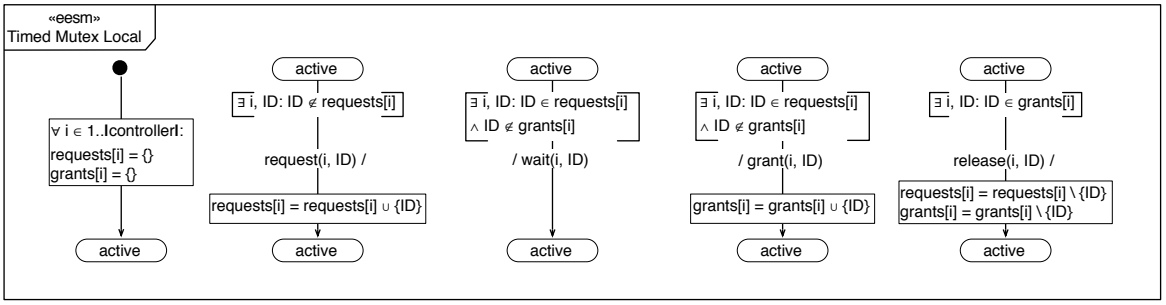

Figure 4. EESM of the Timed Mutex Local block

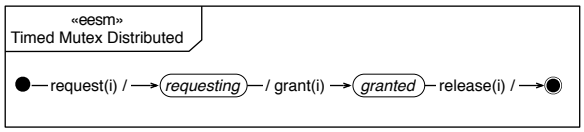

Figure 5. EESM of the Timed Mutex Distributed block

a grant. This will cause the token to be sent towards the controller component and received in a later step to enter *tml:release(ID)*, resetting its EESM with respect to that ID.

The activity of the Timed Mutex Local block is shown in Fig. 6. The protocol to ensure mutual exclusion is implemented by a combination of two blocks: The block of type Mutex ensures that only one locker is given read permission at a time and keeps track of the lockers that have requested permission. The Wait block is taking care of the notification towards each individual locker. It is instantiated as a multi-session block, which means that it is executed with several instances, one for each locker ID, signified by the additional shadow around it and the parameter *(n)*. This session pattern simplifies the modelling of concurrent behaviour, since each session instance only has to keep track of the protocol state of a single locker.

Due to space constraints, we do not show the contracts of Mutex and Wait, but give an informal overview of the behaviour of Timed Mutex Local: The first request is granted right away, whereas subsequent requests are queued at the Mutex block while the corresponding Wait block instances periodically send out tokens via their *keepWaiting* pins. This period is shorter than the duration of the Timer block in Fig. 2, to prevent its expiration under non-failure conditions. Once a previously granted locker sends a release, or the Wait block instance for this locker times out, the next ID in the Mutex queue is used to tell its Wait block instance to grant read permission. Hence, timeouts from the Wait block instances ensure that the protocol can continue even if the release of a read permission is never received.

As there is more than one instance of the Wait block, we have to use select statements when communicating with them. More importantly, since each Wait block instance is implemented as its own state machine, the run-time-support system treats messaging between the parent state machine (the controller) and the children state machines (the Wait block instances) in an asynchronous manner, but with the FIFO property.

3.2 Analysis A_2 of Timed Mutex Local

To verify more detailed properties of our specifications than analysis A_1 supports, we use a formalism based on temporal logic, the Temporal Logic of Actions (TLA, [20]). We can generate TLA$^+$, the language for TLA, automatically from the Arctis models [18],

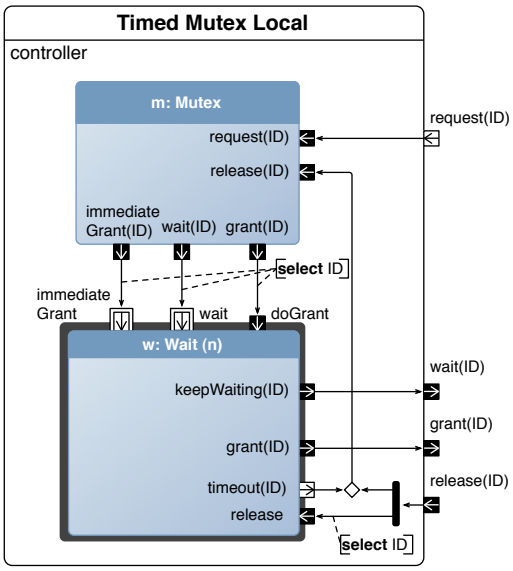

Figure 6. UML activity showing the part of the mutual exclusion protocol that is local to the controller

although not all features introduced in this paper are yet supported by the implementation.

Figure 7 shows an excerpt of the TLA$^+$ specification of the EESM, the activity and the consistency proof for Timed Mutex Local, focusing on the events related to a token passing through the *request(ID)* and *wait(ID)* pins. Each run-to-completion step of the activity, and each EESM transition, is represented as one TLA$^+$ action.[3] The TLA$^+$ actions for the EESM transitions are quite similar to the graphical representation. The main difference is that updates of variables are written in a different style and that variables that are not changed in a transition are explicitly marked as such. The activity part of Fig. 7 refers to contracts of inner blocks by $\langle$*block name*$\rangle$*!*$\langle$*(E)ESM transition*$\rangle$ like *m!request_wait(1, ID)*.[4] It also uses functions *sendToWait(*$\langle$*pin name*$\rangle$*, ID)* and *receiveFromWait(*$\langle$*pin name*$\rangle$*, ID)*, which are both functions we have defined to asynchronously send and receive tokens with the given ID via the named pins. From the part of the specifica-

[3] In TLA and TLA$^+$, an action is a predicate on a pair of system states, modelling the changes to the variables that are carried out in a system step.

[4] The index parameter of the Mutex block is hard coded as 1, since there is only one instance being used in the activity. All EESMs still have an index, so we can use the same TLA$^+$ segment to express them regardless of how many instances are actually used.

```
MODULE Excerpt_for_request_and_wait
From the EESM
request(i, ID) ≜
∧ ID ∉ requests[i]
∧ requests' = [requests EXCEPT ![i] = requests[i] ∪ {ID}]
∧ UNCHANGED ⟨grants⟩

wait(i, ID) ≜
∧ ID ∈ requests[i]
∧ ID ∉ grants[i]
∧ UNCHANGED ⟨grants, requests⟩

From the activity
request_m_request_m_wait_w_wait(ID) ≜
∧ m ! request_wait(1, ID)
∧ sendToWait(waitPin, ID)
∧ UNCHANGED ⟨w_state, fromWait⟩

request_m_request_m_immGrant_w_immGrant(ID) ≜
∧ m ! request_immediateGrant(1, ID)
∧ sendToWait(immediateGrantPin, ID)
∧ UNCHANGED ⟨w_state, fromWait⟩

w_keepWaiting_wait(ID) ≜
∧ receiveFromWait(keepWaitingPin, ID)
∧ UNCHANGED ⟨w_state, m_queue, toWait⟩

From the proof
requestEvent(ID) ≜
∧ eesm ! request(1, ID)
∧
   ∨ act ! request_m_request_m_wait_w_wait(ID)
   ∨ act ! request_m_request_m_immGrant_w_immGrant(ID)

waitEvent(ID) ≜
∧ act ! w_keepWaiting_wait(ID)
∧ IF ENABLED eesm ! wait(1, ID)
   THEN eesm ! wait(1, ID)
   ELSE UNCHANGED ⟨requests, grants⟩

p1 ≜ □(∀ ID ∈ IDs :
ENABLED eesm ! request(1, ID) ⇒
   (ENABLED act ! request_m_request_m_wait_w_wait(ID)
   ∨ ENABLED
      act ! request_m_request_m_immGrant_w_immGrant(ID)))

p2 ≜ □(∀ ID ∈ IDs :
ENABLED act ! w_keepWaiting_wait(ID) ⇒
   ENABLED eesm ! wait(1, ID))
```

Figure 7. TLA$^+$ excerpt for the proof of the Timed Mutex Local block

tion headed by "From the proof", we see that the events of the EESM and the activity are connected so as to take place together in one atomic step. For example in the action *requestEvent(ID)*, whenever a *request(i, ID)* action takes place for the EESM, either a request *m_request_m_wait_w_wait(ID)* or *request_m_request_m_immGrant_w_immGrant(ID)* action must take place in the activity at the same time. In contrast, as modeled by *waitEvent(ID)*, when the activity wants to send a token through the wait pin, we may accept that the EESM does not allow it, due to the semantics of discarding tokens that attempt to travel via pins at the wrong time. This is expressed with the IF-THEN-ELSE construct.

Theorems that verify the consistency of the activity and its (E)ESM can be generated automatically. For example, to verify that a token accepted through pin *request* by the EESM is also accepted by the activity, we use the model checker TLC [29] to check the invariant *p1* from Fig. 7. The invariant states that whenever the EESM is ready to allow a token through the *request* pin (i.e., the corresponding TLA$^+$ action is enabled), one of the corresponding transitions of the activity are also enabled. Invariant *p2* states the same thing for events where tokens pass through the outgoing wait pin. The difference is that any violation of this invariant is interpreted as just a warning, not something that necessarily has to be corrected. Instead, the developer should make sure that any scenarios where the activity can send a token through the pin, but the EESM does not allow it, are intentional. The scenarios are given automatically by the model checker in the form of an error trace that we can visualize in the Arctis models [18].

The design of Timed Mutex Local shown in Fig. 6 has a bug that can lead to a deadlock situation. When running TLC, it returns an error trace to show that the design allows the following scenario:

1. A request from locker L1 is received, granted and released, but the release only gets as far as being duplicated by the fork node so that one token reaches *m:release(ID)* to remove the request entry, while the other token has not yet been received by the Wait block through its *release* pin.
2. The Wait block for locker L1 emits a token through its timeout pin, which rests in the asynchronous channels from the Wait blocks to the controller.
3. Another request from locker L1 is received, an entry is added in the Mutex block and a token put in the queue to the corresponding Wait block via pin *immediateGrant*.
4. The timeout from the L1 instance of Wait is finally received by the controller and reaches *m:release(ID)*, which removes the new request from locker L1 instead of the original request.

This causes an inconsistency between the state of the Wait blocks and the state of Mutex that later on can deadlock the system. The chance of this sequence of events actually happening is very small, especially as the delay from the local, but asynchronous message passing between the Wait blocks and the rest of the controller partition is expected to be much shorter than the non-local message passing. However, these are the kind of subtle faults that could take down a system after years of operation and be very difficult to pinpoint when trying to figure out the reason for the failure.

One could solve the problem by changing the run-time-support system so that local messages always have priority over non-local ones, or we can insert a First block that ensure that either only the release or the timeout for a certain ID reaches Mutex, as shown in Fig. 8. The nice thing about this new version, is that the EESM is exactly the same as before, so the replacement does not trigger a need to redo any of the verification done already. Using the EESM of the Timed Mutex Local block, we just need to verify consistency of the EESM and activity for this new block.

Although we primarily want to prove the BEME property for the Timed Mutex Distributed block, we can also express it for the Timed Mutex Local block. This "BEME Local" property will then be *"if there are ever no more timeouts, then eventually the block will grant at most one read request at a time"*. This is a liveness property, meaning that it describes something that should happen. In our method, the theorems for these properties are formulated manually. To verify liveness properties, we need to add constraints to filter out behaviours that are unreasonable for a real system, such as a dice that is rolled infinitely many times yet never shows a "six". Typically this means constraining the behaviours to those where things that can always or infinitely often happen, do actually happen sometimes, known as *fairness* constraints [20]. To verify the BEME Local property, we also add a liveness constraint to the specification stating that eventually there will never be any more timeouts from any of the Wait block sessions, hence TLC only considers the part of the state space where this holds. We can then express the BEME local property in TLA$^+$

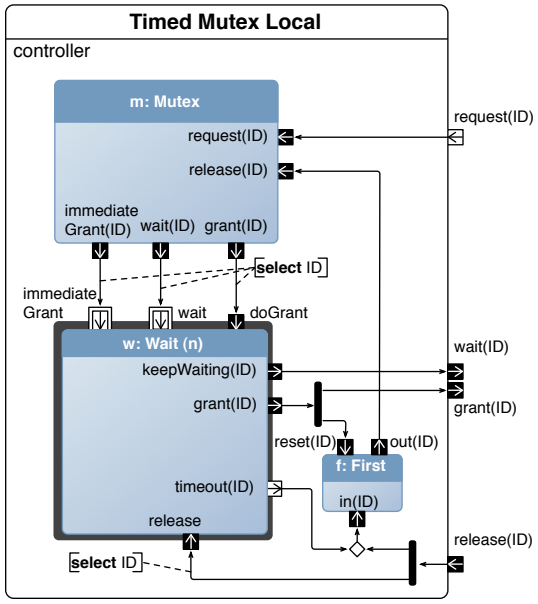

Figure 8. The new and improved activity of Timed Mutex Local

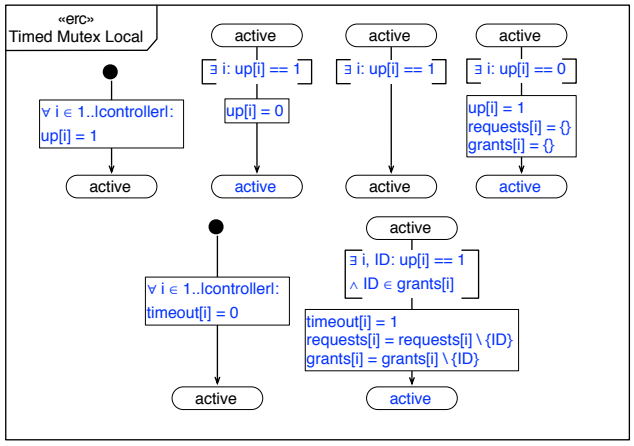

Figure 9. ERC of the Timed Mutex Local block

as $\Diamond\Box(Cardinality(grants[1]) \leq 1)$, which reads *"eventually always the cardinality of the set of granted IDs is at most one."*

Although advanced properties like this one, at least currently, have to be written manually, the reliability expert only has to add two lines to a TLA$^+$ specification of perhaps hundreds of lines to get TLC to check it. This shows that the automation provided by our method can also be very useful for the cases where some things have to be done manually.

We do not have the space to go in detail also about the verification of the Timed Mutex Distributed block. The only truly new property compared to the Timed Mutex Local block, is that we would like to verify that every request received from a locker is eventually responded to via a grant. Again, we will have to add fairness constraints to filter out unrealistic behaviours, and then we can express the property as *"always, for all lockers, a request from a given locker has been received implies that eventually a grant is sent to that locker."*

4. Modelling Realistic Semantics

Our interface contracts help developers to understand the behaviour of a block and facilitate compositional verification by describing only the events that are visible to the outside of the block. In order to do compositional verification under realistic semantics, the contracts must therefore describe the externally visible effects of process crashes and message loss, too. This could be done by writing a new ESM or EESM that includes extra transitions caused by these types of events, with the drawback of having to maintain two, partially overlapping, contracts, one for ideal semantics and another one for realistic semantics. To avoid this potential for inconsistency, we use an aspect-oriented notation to express what we call External Reliability Contracts (ERCs).

The ERC for the Timed Mutex Local block is shown in Fig. 9. In the following, we use the terminology of aspect-oriented programming, AspectJ in particular [1]. The part of the ERC in black is the *pointcut*, the pattern that must match in order to insert the *advice* given in blue colour. The first transition has a pointcut that looks for any transition that starts in an initial state and ends in a state called *active*. Looking at the EESM in Fig. 4, we see that there is only one such transition. Hence, the ERC adds an extra variable *up* (an array, in case there is more than one instance of the block) to the EESM to denote the state of the process running the controller partition. The ERC transition to the right of this matches any state named *active* and adds a new transition with target state *active* so that every controller instance that is not crashed, i.e., up[i] = 1, can crash. The following transition matches any EESM transition that has state *active* as both its target and source. It adds an additional guard stating that the controller must be up in order for such an EESM transition to take place.

As shown in the top right corner, the ERC adds a transition from state *active* with *up* equal to 0 that restarts the controller. A transition with this guard tells us whether the existing state of the EESM is remembered after a crash or not. There is no persistent storage for the state of this block, so both the set of requests and grants are reset to the empty set upon restarting the component. Exchanging the Timed Mutex Local block and its ERC for an otherwise identical block that does use persistent storage, could alter the results of verification with realistic semantics. All local blocks have an ERC with at least these four transitions.

The timeout of a Wait block instance is not explicitly expressed in the EESM of Timed Mutex Local, as it does not trigger any tokens to traverse any external pins. However, the application-specific BEME property of Timed Mutex Distributed is conditional in timeouts eventually not happening anymore, so we want to export this event to the enclosing level through the ERC. As a result we include two more ERC transitions that are specific to exporting the timeout event, placed below the first row of transitions. More precisely, they export the activity step where a token arrives at the controller from *w:timeout(ID)* and reaches *m:release(ID)*, i.e., the event that can cause a new locker to be granted read permission before the previous holder has released it. The first transition adds a timeout variable to the contract, which is initialized to 0. The second transition states that a timeout for a given ID may happen when the component is not crashed and the ID is an element of the set of grants. It also removes the ID of the timed out Wait block instance from the contract sets, to allow new requests even if the release message is lost. Once the timeout event is explicit in the contract like this, we can use it to verify the BEME property for the Timed Mutex Distributed block.[5]

[5] We can also verify the BEME property compositionally under ideal semantics, by first verifying that Timed Mutex Local only gives two or more grants at the same time if a timeout has happened, and then we can refer to

Note that while the EESM is constructed independently from the ERC, the opposite is not true. The ERC is tailored to the existing EESM, and may have to change if the EESM changes.

We skip showing the ERC of Timed Mutex Distributed. Since each locker has a timer that effectively masks the effects of message loss or the controller crashing, the ERC is as simple as the first row of transitions from the ERC in Fig. 9. Not surprisingly, we find this pattern in other blocks as well: Blocks that are built to tolerate failures under realistic semantics have almost the same contract under ideal semantics; the ERC only adds that tokens cannot pass through pins of partitions that have crashed.

In principle, one can match different ERC transitions to the same (E)ESM transition, so that the application of one ERC transition un-matches others. Because of this, we currently demand that any ERC is expressed so as not to have conflicting aspect transitions, meaning that they can be applied in any order and get the same result. We also do not apply ERC transitions to (E)ESM transitions created by other ERC transitions, hence any such transitions must be self-contained, like the last transition in Fig. 9 that already includes "up[i] == 1" since this will not be added by the third ERC transition.

As seen in the next section, realistic semantics greatly increase the size of the state space. Having ERCs as aspects to (E)ESMs means that we always have access to a less complex specification with ideal semantics as well, easily trading details in the behaviour for verifying with more component instances when needed.

5. Discussion

To evaluate the usefulness of the EESMs and ERCs in terms of enabling compositional verification, we observed model checking runs of different specifications. Our hypothesis is that compositional verification will greatly reduce the verification effort, both when considering only ideal semantics (EESMs) and when checking with full realistic semantics (EESMs + ERCs). Table 1 gives the number of states found and the time it took to search through them for several variants of the Timed Mutex Distributed block and the Timed Mutex Local block.[6] The rows of the table are as follows:

Number of lockers → / Alternative ↓		1	2	3
Ideal Semantics				
TML	*states*	20	1 441	125 648
	time	0 sec	4 sec	56 sec
TMD	*states*	54	2 961	157 464
	time	0 sec	3 sec	44 sec
TMD Comp.	*states*	74	4 402	283 112
	time	0 sec	7 sec	100 sec
TMD Direct	*states*	166	388 018	> 80 M
	states, x TMD Comp	2.2 x	89 x	> 282 x
	time	3 sec	222 sec	> 16 hours
	time, x TMD Comp	-	32 x	> 576 x
TMDD NC	*states*	8 746	> 193 M	-
	states, x TMD Comp	118 x	> 44 296 x	-
	time	4 sec	> 29 hours	-
	time, x TMD Comp	-	> 14 914 x	-
Realistic Semantics				
TML	*states*	40	5 636	500 408
	time	1 sec	5 sec	359 sec
TMD	*states*	1 152	331 776	95 551 488
	time	3 sec	98 sec	64 101 sec
TMD Comp.	*states*	1 192	337 412	96 051 896
	time	4 sec	103 sec	64 460 sec
TMD Direct	*states*	1 920	25 970 688	-
	states, x TMD Comp	1.6 x	77 x	-
	time	4 sec	27 320 sec	-
	time, x TMD Comp	1 x	123 x	-

Table 1. Number of states and time to find them using TLC

TML gives the results for model checking the Timed Mutex Local block from Fig. 6 compositionally. That is, using the contracts of the inner blocks, Mutex and Wait, to abstract them.

TMD gives the results for model checking Timed Mutex Distributed from Fig. 2 when abstracting Timed Mutex Local by its contract.

TMD Comp. gives the aggregate results for compositional verification of Timed Mutex Distributed. This row is simply the sum of the results from the two rows above it. We use these number as a base line to compare the other numbers to.

TMD Direct represents monolithic verification and gives the results for model checking the Timed Mutex Distributed block and the Timed Mutex Local block at the same time. Here, we keep the EESM of Timed Mutex Local to filter out tokens traversing the boundaries of the block. This is to demonstrate the effect of analysing both parts at once, instead of separately. The numbers with unit *x* give how many times larger the number of states or seconds is compared to the base line of TMD Comp.

For one locker, TMD Direct has just over double the state space of TMD Comp., and for two lockers the monolithic verification needs 89 times the state space and takes 32 times as long as compositional verification does. For three lockers, the difference is even greater: More than 282 times the state space and more than 576 times the time is used for monolithic verification. All numbers in the table prefixed by ">" are just an indication of the lower bound, as we terminated the model checking run at that point. Note that monolithic verification here only refers to removing one layer of abstraction, the Timed Mutex Local block, not replacing all blocks in Timed Mutex Distributed by their inner contents.

TMDD NC shows the results when attempting to model check the Timed Mutex Distributed block without any contract between it and the contents of the Timed Mutex Local block, i.e., simply adding the contents from the Timed Mutex Local block from Fig. 8 into Fig. 2. In this case, we do not have the contract of Timed Mutex Local to filter out behaviour between the part that came from Timed Mutex Local and the part from Timed Mutex Distributed. A comparison with the compositional case is thus not correct from the point of view of verification: Such a block not only needs to be verified in one run, but also passes more tokens between its parts, increasing the number of possible behaviours in each part. It is simply not the same specification. Nevertheless, we include it by the row TMDD NC (No Contract) to show the practical result of attempting to build the Timed Mutex Distributed block in this way. The results show that we can only model check the specification for one locker within reasonable time.

Considering realistic semantics means considering a much bigger state space, hence compositional verification is necessary even for small models. To do compositional verification in this case, we

whether the set of grants has had more than one element, instead of referring directly to whether a timeout has happened.

[6] We use the variant of Timed Mutex Local with the First block as that allows to search the whole state space without encountering a deadlock.

incorporate the ERCs into the contracts. The results of the model checking are as follows:

TML When analysing under realistic semantics, the state space of Timed Mutex Local almost quadruple for more than one locker. This is as expected since there are two new Boolean variables, *up* and *timeout*, to keep track of. We only track timeouts that allow another locker to get the read permission, hence this variable is never changed for the case with only one locker.

TMD It gets more complex for the Timed Muted Distributed block, as it contains message channels that can drop messages. Since it has more than one partition type and can have more than one instance of the locker partition, there are many combinations of crashes possible. Together with message loss, this greatly increases the state space under realistic semantics.

TMD Direct Just like with ideal semantics, we see an increase in the state space when model checking with both parts of the system at once with realistic semantics. The difference is that the numbers for the compositional verification are already much higher under realistic semantics, so the exponential blowup due to monolithic verification has a much greater effect in practise. This is especially so for the time taken to search the state space, as TLC tends to search fewer states per second for larger state spaces.

As we can see, the number of lockers contributes to the state space in an exponential manner, when not using any state space reduction techniques.[7] Hence, it is important to keep the starting numbers low, so that we can verify the specifications for a large enough number of lockers that we gain confidence in their correctness.[8] This is where the benefit of our compositional approach comes in: By analysing tightly coupled parts of the behaviour while abstracting the rest, we can avoid the exponential growth in the state space that stems from analysing too many parts of the behaviour at once. This in turn, allows us to reach further with respect to the number of instances of the same type the model checker can handle. To sum up, we see that compositional verification performs significantly better than monolithic verification with contracts, and that trying to build the system without contracts most likely would lead to a state space that is infeasible to model check at all with multiple lockers.

6. Related Work

The idea of compositional verification of temporal logic specifications, whether it is by manual proofs [13] or model checking [7], is not a new one. There are also several approaches that automate the compositional verification process. However, it is not trivial how a system is decomposed. Cobleigh et al. [9] use the L* learning algorithm coupled with a model checker for automatic assume–guarantee reasoning [8] about the properties of systems. They report that "the vast majority" of the 2-way decompositions found for each example system actually did not improve on monolithic verification. In fact, their results were not very promising: Only about half of the examples studied could be improved by assume–guarantee reasoning, and even in these cases the gains were mostly limited to expanding the model by one instance. However, later studies on the topic report better results [5, 23], although the improvements over monolithic verification are seldom by more than factor 4. So why are our experimental results so different? Although these works and ours both deal with compositional verification, they are not directly comparable: First of all, we report on a single, albeit real, system. This is not enough to say anything precise about the performance. Further, these works find a new assumption for each property to verify, while we use a static contract for all properties. While they are looking for an assumption that perfectly abstracts the other part of the system for a given property, we take advantage of the fact that we control the resulting implementation and carry any extra constraints from the contracts into the actual implementation. Also, our development method naturally leads to tightly coupled clusters of behavioural logic with looser coupling between them, due to the inherent goal of creating reusable building blocks.

We use UML activities to model software components. There are other works giving UML activities a formal semantics [10, 11, 28], but these all omit contracts to enabled hierarchical activities. As pointed out in [4], we can only expect software components to be reused for critical systems if they come with clear instructions on how to be correctly reused and what guarantees they give under those conditions. UML already provides the concept of Protocol State Machines [22] to detail how a component can communicate with its environment. Mencl [21] proposes Port State Machines to improve on several shortcomings of Protocol State Machines, for example that they do no allow nesting or interleaving method calls, nor dependencies between a provided and required interface. His Port State Machines split method calls into atomic request and response events to allow for nesting and interleaving method calls, but they are restricted to pure control flow, as transition guards are not supported. Bauer and Hennicker [3] introduce a protocol description that is a hybrid of control flow and data state styles. However, this approach also lacks the ability to express dependencies between required and provided interfaces.

Like Port State Machines, our contracts have atomic interface events to allow for the expression of nesting and interleaving method calls. As they abstract both the block and its environment, they also express the provided and required interfaces in the same structure, hence allowing to express dependencies between them. In addition, our EESMs combine this with data variables so that we can more accurately express the behaviour of blocks with many instances [26] or blocks whose behaviour is otherwise strongly data dependent.

Sanders et al. [25] present semantic interfaces of service components, using finite state machine notation to describe both. Semantic interfaces can be used to find both complementary and implementing components, hence they support compositional verification. The main difference from our contracts is that semantic interfaces abstract local components that are asynchronously connected to remote or local components, while our contracts are mainly used internally in one process to connect sub-components, described as activity diagrams, synchronously together.[9] The fact that our contracts allow encapsulation of both local components and distributed collaborations between components, sets them apart from all the above.

7. Concluding Remarks

While we currently either analyse a specification under completely ideal or realistic semantics, we see an advantage in having more fine-grained control over the execution semantics of individual system parts. This could be achieved by extending our method and tool with a deployment model where one could easily alter which components and channels should have ideal or realistic semantics,

[7] We present data from unoptimized specifications, as the main point is to show the relationship between numbers of different rows.

[8] Model checking cannot prove properties of general models, only the model instances that are actually checked. Hence, there could be a number of lockers for which the properties do not hold. The best we can do is to check with a few instances for which most bugs will manifest themselves.

[9] As seen from the Wait blocks in Fig. 6, asynchronous coupling is also supported.

or even the ordering properties of a channel. For the scenario above, such a tool would enable faster analysis for one failure source at a time, but not reveal any problems caused by a combination of failure sources.

All ERCs written for this case study have been made manually. However, we could automate most of their construction for local blocks, which all have the three first transitions from Fig. 9 in common. If we enable developers to tag elements of the existing model with a ≪persistent≫ stereotype, we should be able to automatically generate the restart transition as well.

ERCs are used to export reliability-related events that are not directly visible as tokens passing through pins. It may be that other non-functional properties of our models can be described in the same manner, adding them as aspects to (E)ESMs. For example, there is work to analyse security aspects of SPACE models [12], and we can imagine a use for a concept like this to export security aspects from inside a block to a higher level.

If we are to apply several aspect-oriented contracts to each base contract, perhaps even created by different people, the problem of conflicting aspects transitions is likely to increase. In such a case, we might need to develop a conflict resolution mechanism to ensure that there is no ambiguity in the result of the aspect weaving.

In summary, we have shown that encapsulation using our contracts can reduce the state space to verify by at least factor 100 compared to monolithic verification. We have introduced ERCs to allow compositional verification also under realistic semantics. ERCs allow to easily switch between analysis under ideal or realistic semantics. Since the state space under realistic semantics is larger than under ideal semantics, this allows trading realistic behaviour descriptions for larger model sizes, when convenient.

References

[1] ApectJ web site. URL `http://www.eclipse.org/aspectj`. Last accessed May 2011.

[2] Arctis Verification Project. Norwegian Research Council, FORNY Project no. 199644.

[3] S. Bauer and R. Hennicker. Views on Behaviour Protocols and Their Semantic Foundation. In *Algebra and Coalgebra in Computer Science*, volume 5728 of *LNCS*, pages 367–382. Springer, 2009.

[4] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32:38–45, 1999.

[5] Y.-F. Chen, E. M. Clarke, A. Farzan, F. He, M.-H. Tsai, Y.-K. Tsay, B.-Y. Wang, and L. Zhu. Comparing Learning Algorithms in Automated Assume-Guarantee Reasoning. In *Proc. of the 4th int. conf. on Leveraging applications of formal methods, verification, and validation - Volume Part I*, ISoLA'10, pages 643–657. Springer-Verlag, 2010.

[6] K. T. Cheng and A. S. Krishnakumar. Automatic Functional Test Generation using the Extended Finite State Machine Model. In *Proc. 30th Int. Design Automation Conf.*, DAC'93, pages 86–91. ACM, 1993.

[7] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *Proc. of the Fourth Annual Symposium on Logic in computer science*, pages 353–362. IEEE Press, 1989.

[8] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proc. of the 9th int. conf. on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 331–346. Springer-Verlag, 2003.

[9] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking Up is Hard to Do: An Investigation of Decomposition for Assume-Guarantee Reasoning. In *Proc. of the 2006 int. symposium on Software testing and analysis*, ISSTA '06, pages 97–108. ACM, 2006.

[10] R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[11] N. Guelfi and A. Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *Proc. 12th Asia-Pacific SE Conf.*, pages 283–290, 2005.

[12] L. Gunawan, F. Kraemer, and P. Herrmann. A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications. In *Engineering Secure Software and Systems*, volume 6542 of *LNCS*, pages 142–155. Springer Berlin / Heidelberg, 2011.

[13] P. Herrmann and H. Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[14] F. A. Kraemer. *Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. PhD thesis, Norwegian University of Science and Technology, 2008.

[15] F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In *Proc. of the 12th Int. Conf. on Model Driven Engineering, Languages and Systems (Models)*, volume 5795 of *LNCS*, pages 571–585, 2009.

[16] F. A. Kraemer and P. Herrmann. Reactive Semantics for Distributed UML Activities. In *Formal Techniques for Distributed Systems*, volume 6117 of *LNCS*, pages 17–31, 2010.

[17] F. A. Kraemer, R. Bræk, and P. Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In *Proc. 13th Int. SDL Forum Conf. on Design for Dependable Systems*, SDL'07, pages 166–185, 2007.

[18] F. A. Kraemer, V. Slåtten, and P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *Journal of Systems and Software*, 82(12):2068–2080, 2009.

[19] F. A. Kraemer, V. Slåtten, and P. Herrmann. Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example. In *SDL 2009*, volume 5719 of *LNCS*, pages 1–18. Springer-Verlag Berlin Heidelberg, 2009.

[20] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[21] V. Mencl. Specifying Component Behavior with Port State Machines. *Electronic Notes in Theoretical Computer Science*, 101:129–153, 2004.

[22] Object Management Group (OMG). Unified Modeling Language: Superstructure, Version 2.3, 2010.

[23] C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning. *Form. Methods Syst. Des.*, 32:175–205, 2008.

[24] J. Rushby. Disappearing Formal Methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96. ACM, 2000.

[25] R. T. Sanders, R. Bræk, G. von Bochmann, and D. Amyot. Service Discovery and Component Reuse with Semantic Interfaces. In *SDL 2005: Model Driven Systems Design*, volume 3530 of *LNCS*, chapter 6, pages 1244–1247. Springer, 2005.

[26] V. Slåtten and P. Herrmann. Contracts for Multi-instance UML Activities. In *Formal Techniques for Distributed Systems*, volume 6722 of *LNCS*, pages 304–318, 2011.

[27] V. Slåtten, F. A. Kraemer, and P. Herrmann. Towards a Model-Driven Method for Reliable Applications: From Ideal to Realistic Transmission Semantics. In *Proc. 2nd Int. Workshop on Software Engineering for Resilient Systems (SERENE 2010)*. ACM Digital Library, 2010.

[28] H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theor. Comp. Sci.*, 127(4):35–52, 2005.

[29] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In *Proc. 10th IFIP WG 10.5 Adv. Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, 1999.