Formalizing Collaboration-Oriented Service Specifications using Temporal Logic

Frank Alexander Kraemer and Peter Herrmann
Department of Telematics
Norwegian University of Science and Technology (NTNU)
7491 Trondheim, Norway
{kraemer, herrmann}@item.ntnu.no

Abstract

In our highly automated engineering approach, reactive services are specified using UML 2.0 collaborations and activities. This enables to focus on complete behaviors between of a set of participants in isolation, and to decompose systems according to the functionalities it should offer. Of course, precise semantics for the specifications are necessary, as we use them as input for model checking and automatic synthesis of components for implementation. For this reason we formalize the concept of collaborations in the temporal logic cTLA by defining the specification style cTLA/c. Collaborations are hereby represented as cTLA processes, and the composition of collaboration can be reduced to process couplings. While cTLA/c is general to capture the semantics of different languages, we show in detail how UML 2.0 activities are mapped to cTLA/c by a set of cTLA processes and production rules.

1 Introduction

A networked service is a system offering certain functionalities that are used by concurrently acting entities in its environment. The service functions often render a reactive behavior in the sense that they *"maintain some interaction with their environment"* [25]. From a physical point of view, such a system naturally decomposes into its components, that means the distributed entities providing the system functionality. In the setting of a model-driven development approach, the components may be expressed for example by SDL processes and blocks [11], or UML state machines and composite structures [24]. These component descriptions form the input for automatic code generation tools (see, e.g., [3, 7, 30]).

For a system offering a whole bunch of functionalities to its environment, such a component-based view leads to complex specifications as each component model describes partial aspects of various functionalities. Instead, we desire a specification style in which a specification block models all aspects of a single functionality facilitating the individual development, deployment, invocation and maintenance of separate functionalities. As a functionality is basically a service spanning over several components, we therefore need specifications describing the collaboration of various components. Modeling languages like UML interactions [24], MSC [12] and Use Case Maps [4] offer a solution by enabling the description of both partial and collaborative behaviors. We apply UML 2.0 collaborations to express static properties and UML 2.0 activities to model collaborative behavior [9, 16, 17].

Of course, the need for component models remains as, in the end, the components are the entities which have to be created and deployed on different devices to realize the system. Consequently, we often find system models utilizing diagrams of several types which describe a system from different viewpoints. This, however, imposes the challenge of keeping the diagrams consistent. Given the need to build services and to adjust their system functions rapidly, approaches that rely on the discipline of its developers to maintain the diagrams manually are rather naive. A consequent way to go is therefore to let developers create only one group of diagrams and infer the others completely automatically, for example by means of model transformations.

Several examples [5, 9, 16, 17, 26, 28] illustrate how the notion of collaborations can be used to specify services. All these specifications have in common that they decompose a system according to its tasks, delay the construction of components to a later stage, and identify only participants relevant for the modeled functionality. Such tasks (or sub-functionalities resp. sub-services) often show up in more than one application. They typically have a concise objective or function, resulting in building blocks which can be used for various service descriptions in a particular application domain.

With our approach for the specification by activities, collaborations

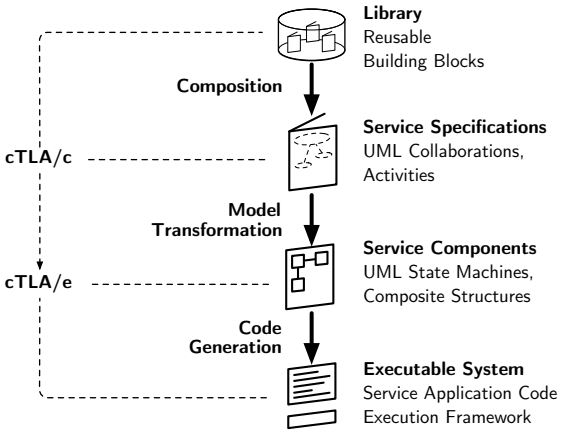


Figure 1: The SPACE Engineering Approach

and executable state machines (SPACE), we develop a tool-supported process that analyzes and transforms collaborative service specifications in a highly automated way [17]. The approach is outlined in Fig. 1. Often, a new service can be composed from already existing collaboration patterns that may be adapted to the operations and data types specific for the application under construction. An engineer therefore may consider a library of reusable building blocks which are subsequently composed obtaining a composite service specification. To come to an executable system, the service specifications in terms of UML 2.0 activities and collaborations are then transformed into UML 2.0 components and state machines, as detailed in [19]. From this representation, code generation is quite straightforward, as for example explained in [20]. In this way, the engineer just works on the creation of the service specifications, while the rest of the approach is automated, so that consistency is ensured by construction. Tool support for the SPACE approach exists in form of two integrated Eclipse-based tool sets [14]. *Arctis* offers support for collaborative service specifications, their analysis and transformation into executable state machines. These can then be further analyzed by *Ramses*, which also offers code generators to create implementations based on Java.

Of course, to guarantee the precise understanding of the models and the correctness of the transformations, the approach requires formal reasoning on the semantics of the languages used, the transformation tools, and the consistency of building blocks and their composition. Temporal logic is a suitable instrument for that. In particular, the principle of superposition supported by cTLA [8, 10] makes it possible to describe systems from different viewpoints by individual processes that are superimposed (see Sect. 3.2). Therefore, the development approach in Fig. 1 is complemented by formal reasoning (shown on the left side). In a first step, we formalized the behavior of the state machines with cTLA/e [20]. This cTLA style defines a set of constraints for cTLA specifications that directly reflect the special properties of the state machines needed to enable the generation of efficiently executable code. Due to this foundation of the state machines, we can ensure that the generated program code is compliant with the behavior described in the state machines (see [20]).

In this paper, we focus on the definition of cTLA/c, a style of cTLA that allows us to formalize the collaborative service specifications given by UML 2.0 activities. By expressing collaborations as cTLA processes, we can ensure that a composed service maintains the properties of the individual collaborations it is composed from. The semantic definition presented here enables us to prove formally that the transformation from activities to state machines is correctness-preserving. As sketched in [19], this corresponds to substitute that an activity modeled by the cTLA/c specification A is always transformed to a state machine described by cTLA/e model S which is a correct refinement of A (i.e., $S \Rightarrow A$ holds).

The semantic definition of collaborations and activities in form of temporal logic is implemented as a transformation tool [29] which produces TLA$^+$ modules from activities. These modules may then be used as input for the model checker TLC [32]. The tool also generates a number of theorems, so that collaborations may be analyzed for more advanced properties than simple syntactic checks allow.

In Sect. 2, we introduce service specifications based on collaborations and activities by means of an example. An introduction to general cTLA in Sect. 3 is followed by the presentation of the specification style cTLA/c in Sect. 4. Thereafter, Sect. 5 is devoted to the formal definition

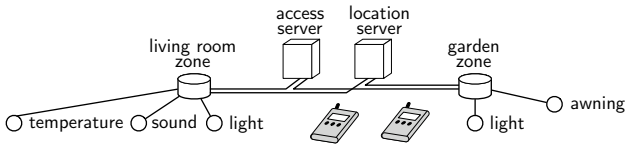


Figure 2: Illustration for a system configuration

of the UML 2.0 activities used in our approach. For that, special cTLA specification blocks to model the behavioral features of the activity nodes are used. In addition, we define a set of production rules guiding the creation of the final cTLA/c specifications according to the nodes and edges of an activity. In Sect. 6, we discuss how these specifications modeling elementary service collaborations can be composed to specify also composite specifications and service descriptions that may handle several users and multiple sessions at a time.

2 Specifications in SPACE

As an example, we consider a system for home automation, which allows the residents of a house to control various devices with their mobile phones. Devices may be heaters and volume controls, lights, motors of awnings, or the intercom with the door bell. The house is organized in zones covering different possibly overlapping areas. Each device is assigned to a zone manager, as illustrated in Fig. 2 with a zone for the living room and one for the garden. To make the user interface easier, the control options offered by a mobile phone should depend on its current location, so that one may adjust the room temperature of only the room one currently stays in, or roll out the awnings when one is on the terrace. For this reason, we assume that the position of the phones can be determined by a location server with sufficient accuracy (e.g., by equipping them with WLAN capabilities). Via this channel, the mobile phones may also communicate to the zone managers. In addition to the location server, an access server keeps record of the user authorizations and access rights, for example, to grant guests and children only limited control.

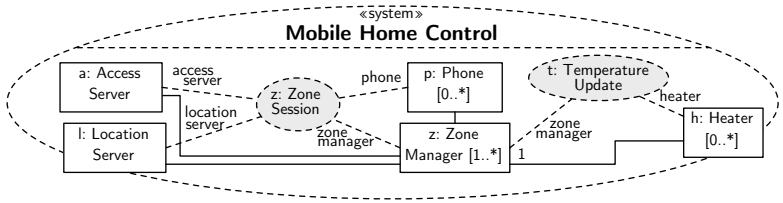


Figure 3: System collaboration

2.1 UML 2.0 Collaborations

Figure 3 shows a UML 2.0 collaboration specifying the structure of the system. For each participant it declares a collaboration role, i.e., a and l for the access and location server, p representing the phones, z for the zone managers and h for a set of heaters[1]. With the stereotype ≪system≫ we express that Fig. 3 documents the highest system level, and that its collaboration roles should be realized as separate components[2]. In addition to the type information, the collaboration roles specify the multiplicity of the components. While access server and location server have default multiplicity "1", there may be any number of phones in the system and an arbitrary number of zones, which in turn may be connected to several heaters. The multiplicity of the connector end right to the zone manager is "1". This tells us that a heater is only connected to one zone manager while one zone manager is connected to many devices.

Collaborations may refer to other collaborations by means of collaboration uses, so that a specification may reuse existing building blocks or be decomposed to reduce complexity. In this way, the specification in Fig. 3 expresses that a zone communicates with the heaters by collaboration *Temperature Update*, which is bound to the system by collaboration use t. Whenever a phone enters a zone, the location server starts a zone session collaboration between the phone and the entered zone, represented by collaboration use z. The labels at the lines which connect the ellipse of the collaboration use describe to which collabo-

[1]To keep the example manageable, we look here only at heaters as devices.

[2]For this reason, we also include the type names (like $ZoneManager$) which will be taken as type names for the components to be generated.

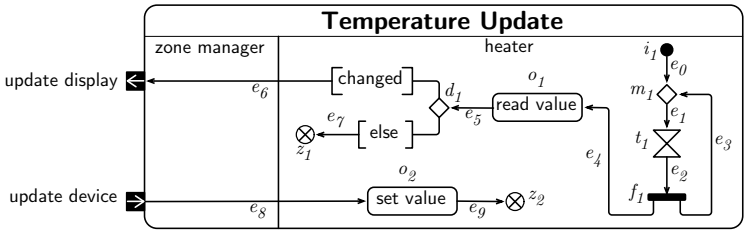


Figure 4: Activity describing the control of the temperature

ration roles of the referring collaboration the collaboration roles of the referred collaboration are bound.

2.2 Activities for Elementary Collaborations

As mentioned in the introduction, SPACE uses activities to describe the behavior of collaborations. Activities can be understood as token flows, similar to Petri nets. Let us first consider the activity for the elementary collaboration given in Fig. 4 for the control of the room temperature with a heater. The activity has two partitions, *zone manager* and *heater*, one for each collaboration role. At startup, a token is emitted from the initial node i_1, which then finds its way through merge node m_1 towards timer t_1, where it starts the timer and rests. When the timer expires, the token is emitted and duplicated within the fork node f_1. One token is then redirected via merge node m_1 back to the timer, which starts again. The other token flows via operation *read value* to decision node d_1. If the temperature changed, the token is sent to the zone manager. For the transmission, we assume a queue place on edges that cross partitions (in the following called *transfer edges*), so that the token first rests within the queue before it is read out by the zone manager. If the change of temperature is insignificant and does not need to be reported to the zone manager, the token flow is ended in the flow final node z_1. A second flow starts at the input parameter node *update device*, forwards to the heater, causes a value change as expressed by the operation o_2, and finally terminates at node z_2.

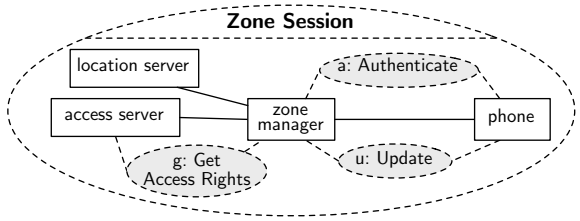


Figure 5: Collaboration for a phone within a zone

2.3 Activities for Compositional Collaborations

The zone session referred from the collaboration in Fig. 3 is further detailed in Fig. 5. It is in turn a composite collaboration with the collaboration roles for the location and access server, a zone manager and a phone as its participants. All collaboration roles have their default multiplicity "1", so that this collaboration describes the cooperation of exactly one phone and one zone manager, covering the behavior whenever a phone is within a certain zone. For that, collaboration uses a, u and g refer to collaboration describing the retrieval of access rights from a server, the authentication of the phone, and the bidirectional update mechanisms between zone and phone. The detailed coupling of these collaboration uses is described by Fig. 6. It has again one activity partition for each of the collaboration roles. In addition, the collaboration uses of Fig. 5 are represented as call behavior actions g, a and u. They refer to other activities that describe their detailed behavior. Their pins are used to couple them together, using some additional logic in the zone manager and the phone. The collaboration starts when the location server detects that a phone enters a zone, via input parameter node *enter*[3]. Upon that, the zone manager simultaneously invokes via the fork node the sub-collaborations g to retrieve the access rights as well as a to request an authentication from the phone. Once the access right information arrives, a token is placed into the waiting decision node. This is an extension of a normal decision node with the difference that a token rests in it until one of the downstream joins may fire [19]. This is the case when the authentication finishes. De-

[3]Activity parameter nodes are represented by pins owned by a call behavior actions when their activity is referred to by another activity.

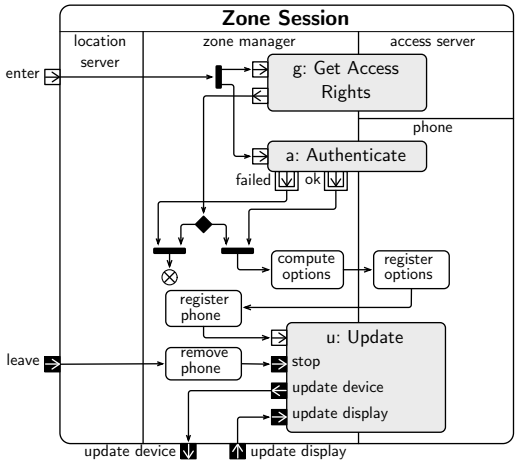


Figure 6: Collaboration for entering and leaving a zone

pending on the outcome, either the left join node may eventually fire, which causes the flow to stop, as the user is not authenticated. Upon a successful authentication, the right join fires and the zone manager computes the options that are offered to the phone. After the phone has registered the options, the zone manager registers the phone enabling it to handle updates. Collaboration u is started which manages the update handling, so that the phone may send updates to the devices and vice versa.

2.4 Multi-Session Collaborations

Within one occurrence of a zone session, each collaboration role was considered only once, and each collaboration use was executed only once at a time. The collaboration for the entire system, *Mobile Home Control* in Fig. 3, however, has several zones, heaters and phones. As a consequence, the collaborations z for the zone sessions and t for the temperature updates are executed with several executions (also called sessions) at the same time. The one location server is invoked in several sessions of a zone session (with different zone managers and phones). One zone manager is connected to many heaters, and maintains there-

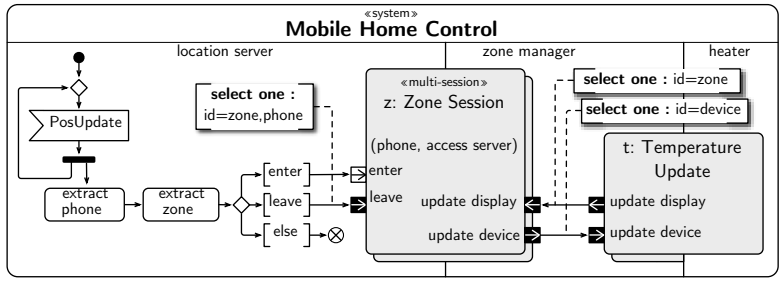


Figure 7: Activity composing the entire system

fore several temperature update collaboration instances with them at the same time. We emphasize this by adding a shadow-like border to the call behavior actions within those activity partitions that can choose from different collaboration instances. In Fig. 7 this is the entire zone session, as both zone manager and location server have to handle several instances. For the temperature update, the collaboration is multiple only within the zone manager (which may be connected to several heaters) but is single within the heater partition, as one heater participates in only one temperature update session as it is connected to only one zone manager. The activity of Fig. 7 starts within the location manager that waits for the reception of position updates. Once a position update arrives, it extracts the phone and zone information from the update and decides, if the change of location should be interpreted as the entering or the leaving of a zone. If the phone enters a zone, a zone session is started, if it leaves a zone, the corresponding ongoing zone session is informed (and stopped). In both cases, the token has to enter a specific zone session instance. UML activities may handle several execution at the same time, that means, a call behavior action may refer to several executions that go on at the same time. However, UML does not provide means to distinguish the different session from each other, so that we may compose them in a more advanced manner. We therefore introduced an selection operator, that distinguishes different session instances by a set of filters (see [16]). To enter the zone session, we take the zone and the phone as ID identifying the corresponding session. More filters that also take data within the session instances into consideration, are described in [16].

3 Temporal Logic and cTLA

Temporal logic enables to specify behavior which, according to Kurki-Suonio, is the *"abstraction of reactive executions"* [21]. Since networked services are reactive by nature, temporal logics are therefore suited to model the service behavior formally. One can distinguish temporal logics in linear time logics (LTL), which express behaviors by sets of infinite sequences of states, and in branching time logics (BTL) modeling state orders by tree structures. While the latter concept offers a higher degree of expressiveness, LTLs often lead to easier understandable specifications.

A well-known LTL is Leslie Lamport's Temporal Logic of Actions (TLA, [23]) in which behavior is described by special state transition systems as well as fairness properties. The TLA coupling method by means of states common to several element specifications [1], however, makes it difficult to create constraint-oriented models in which not single physical components but properties reflecting partial system behavior spanning several components are specified [31]. As our collaboration-oriented models demand exactly this specification style, we use the compositional Temporal Logic of Actions (cTLA, [8, 10]). This is a variant of TLA which provides couplings based on jointly executed transitions enabling to glue interacting constraints nicely. Moreover, cTLA makes the description of state transition systems in a process-like style possible. A cTLA process can either be in a simple form, modeling the state transition systems directly, or in a compositional form combining several process instances which interact by the jointly fired transitions. In the following, we introduce both process types in detail.

3.1 Simple cTLA Processes

Simple cTLA processes are used to model single resources or constraints of a system. Figure 8 depicts a simple process which specifies a timer node of an UML 2.0 activity. In the process header, the process name and a list of process parameters are listed. The parameters enable to model several shapes of process instances by a single process type. For example, the process parameter TT describes the signature[4] of the

[4]Like in colored Petri nets (see [13]), we assume activity tokens to contain special data sets to specify the forwarding of data.

```
PROCESS Timer(TT: Any)
VARIABLES
  i: {"idle", "active"};
  tv:  TT;
INIT ≙ i = "idle" ∧ tv ∈ TT;
ACTIONS
  start(it: TT) ≙
    i = "idle" ∧ i′ = "active" ∧ tv′ = it;
  expire(ot: TT) ≙
    i = "active" ∧ ot = tv ∧ i′ = "idle" ∧ unchanged(tv);
  expireAndRestart(it, ot: TT) ≙
    i = "active" ∧ ot = tv ∧ tv′ = it ∧ unchanged(i);
END
```

Figure 8: cTLA process for Modeling Activity Timers

tokens modeled by a particular UML activity so that we can use the process *Timer* for various token formats. As said, a cTLA process models a state transition system, the state of which is described by variables. In the example process, we use the variables i distinguishing if the timer is idle or active and tv storing the data of an activity token passing it. The set of initial states which hold in the beginning of executing a process are defined by the predicate `INIT`. Here, the variable i is initially idle while tv contains any data set from TT.

The transitions are specified by actions (e.g., *start*) which are predicates on a pair of a current and a next state. Variable identifiers in simple form (e.g., i) refer to the current state while variables describing the successor state occur in the primed form (e.g., i'). The conjuncts of an action referring only to variables in the current state specify the enabling condition while those with primed variable identifiers express the state change. Thus, the action *start* is enabled if variable i is *"idle"* while its execution leads to a new process state change in which i carries the new value *"active"*. Actions may have parameters modeling transfer between processes. For instance, *start* has the parameter *it* of type TT describing the data set of a token arriving at the timer which is stored in the variable *tv*. Actions can be distinguished into two classes. External actions denoted by the keyword `ACTIONS` may be coupled with actions of other processes. In contrast, internal actions defined in a compartment headed with `INTERNAL ACTIONS` must not be joined with actions of the process environment so that they express purely local process behavior.

In the process *Timer* we use only externally visible actions. Moreover, we may provide the actions with fairness assumptions guaranteeing a lively behavior. Since we concentrate in this paper on safety aspects only, we do not discuss that in detail.

Formally, a cTLA process can be expressed as a TLA-formula, the so-called canonical formula C:

$$C \triangleq INIT \ \wedge \ \Box[\,\exists it, ot \in TT : start(it) \ \vee expire(ot) \vee expireAndRestart(it, ot)]_{\langle i,tv\rangle}$$

The conjunct at the left side of the formula states that the predicate `INIT` holds in the first state of every state sequence modeled by C. The conjunct on the right side starts with the temporal operator $\Box$ ("always") specifying that the expression right to it has to hold in all states of all state sequences. The TLA expression $[pp]_{\langle i,tv\rangle}$ defines that either the pair predicate pp holds or that a stuttering step[5] takes place in which the annexed variable identifiers do not change their state (i.e., $i' = i \wedge tv' = tv$ holds). As pair predicate pp we listed the disjunction of the process actions in which the process parameters are existentially quantified. This models that a state change in the process always corresponds to the execution of one of its actions using any action parameters of the set TT. Thus, the cTLA process specifies that the first process state fulfills `INIT` and that all state changes follow the process actions or are stuttering steps.

As outline above, cTLA processes are special TLA formulas which, however, follow certain constraints facilitating the cTLA-based action couplings. Mainly, a process action may access only variables of the process, it is defined in, and, like in DisCo [21], the actions can be uniquely identified which enables a reference of process actions in compositional descriptions. Some other conventions are necessary for guaranteeing liveness properties and are introduced in [8].

3.2 Compositional cTLA Processes

Compositional cTLA processes specify systems and subsystems as compositions of simple cTLA process instances which cooperate by means of synchronously executed process actions. Data transfer between the simple processes is modeled by aligning the parameters of the coupled process actions. Since the variables of the simple processes are encapsulated and cannot be read or modified by other processes, a system state is defined as the vector of the process variables. The system transitions are modeled by the synchronously executed process actions. Each stateful simple process (i.e., each process in which variables are defined) contributes to a system action by either exactly one process action or by a stuttering step modeling concurrency as interleaving. In consequence, a system action is a conjunction of process actions and process

[5]Stuttering steps are necessary for carrying out refinement proofs.

```
PROCESS TemperatureUpdate
  (TT: [[ temp: NATURALS, ANY ]] ;
   VT: [[ tsTemp, tsOldTemp: NATURALS; tsChg: BOOLEAN; ANY ]] )
CONSTANTS
  ET = {"e1", "e2", "e3", "e4", "e5", "e6", "e7", "e8", "e9"};
  nv1 = [n ∈ VT × TT → VT
         ↦ [[n.1 EXCEPT !.tsOldTemp = n.1.tsTemp
                   EXCEPT !.tsChg = n.1.tsTemp ≠ n.1.tsOldTemp];
  nt1 = [n ∈ TT × TT → TT ↦ [[n.2 EXCEPT !.temp = n.1.tsTemp ]] ];
  nv2 = [n ∈ VT × TT → VT ↦ [[n.1 EXCEPT !.tsTemp = n.2.temp ]] ];
  nt2 = [n ∈ TT × TT → TT ↦ n.2];
  gu1 = [n ∈ 1..2 × VT × TT → BOOLEAN
         ↦ IF n.1 = 1 THEN n.2.tsChg ELSE NOT n.2.tsChg ];
PROCESSES
  i1: Initial(TT);
  t1: Timer(TT);
  o1: Operation(nv1,nt1);
  d1: Decision(2,gu1);
  e6: Transfer(TT);
  e8: Transfer(TT);
  o2: Operation(nv2,nt2);
ACTIONS
  act3(it: TT; ot: [ET → TT], iv: VT, ov: [ET → VT],
                is: TT, os: SUBSET TT, last: SUBSET ET) ≜
     i1.start(it)
   ∧ t1.start(it)
   ∧ os = {}
   ∧ ov = ["e0" ↦ iv, "e1" ↦ iv]
   ∧ ot = ["e0" ↦ it, "e1" ↦ it]
   ∧ last = {"e1"}
   ∧ e6.Stutter ∧ e8.Stutter;
  ...
END
```

Figure 9: cTLA Process describing the activity *Temperature Update*

stuttering steps.

Figure 9 models the cTLA specification of the UML activity *Temperature Update* (see Fig. 4) as a compositional cTLA process with the same name. The cTLA process is composed from the process instances listed in the section *PROCESSES*. For example, *t1* is an instance of the process type *Timer* introduced above. The process parameter TT of *t1* is instantiated with the process parameter TT defined as a process parameter in the compositional process. The external and internal system actions are specified in the blocks `ACTIONS` and `INTERNAL ACTIONS` as conjunctions of process actions and process stuttering steps. We depicted the system action act_3 modeling the flow from the initial UML activity node i_1 to the timer t_1. The action is a coupling of the actions *start* in both composed processes *i1* and *t1* while the processes *e6* and *e8* perform stuttering steps[6].

Formally, a compositional cTLA process corresponds to the conjunction of the canonical formulas of the composed simple processes and an additional coupling constraint CC:

$$C \triangleq i1.C \wedge t1.C \wedge \ldots \wedge o2.C \wedge CC$$

The coupling constraint defines the conjunction of the process actions to system actions. Moreover, it defines some constraints on the process action fairness properties which, together with the encapsulation of the process variables, guarantee that the cTLA composition principle fulfills the superposition property (see [8]). Superposition [2] ensures that each property of a simple cTLA process holds also for each compositional one including it. As mentioned in the introduction, this is an important ingredient to describe systems from different viewpoints since we can define elementary service functions as separate simple cTLA processes and compose these easily to both collaborative and component-oriented system models. In addition, this property facilitates the formal proofs vastly that component models realize collaboration-based ones.

A compositional cTLA process can be transformed to an equivalent process in simple form as proven in [10]. Basically, the simple process comprises the local variables of the included process instances as its variable set while the transitions are modeled by the expanded system actions. As an example, Fig. 10 depicts the process *Temperature Update* in simple form. This transformation from compositional to simple cTLA processes is essential for our UML activity modeling approach as we discuss in Sect. 5.

[6]The processes *d1*, *o1* and *o2* are not referred to since they are stateless.

```
PROCESS TemperatureUpdate
  (TT :  [[temp : NATURALS, ANY]] ;
   VT :  [[tsTemp, tsOldTemp : NATURALS; tsChg : BOOLEAN; ANY]] )
CONSTANTS
  ET = {"e1", "e2", "e3", "e4", "e5", "e6", "e7", "e8", "e9"};
VARIABLES
  i1xi : {"init","active"};
  t1xi : {idle, active};
  t1xtv: TT;
  e6xq : QUEUE(TT);
  e8xq : QUEUE(TT);
ACTIONS
  act3(it: TT; ot: [ET → TT], iv: VT, ov: [ET → VT],
                is: TT, os: SUBSET TT, last: SUBSET ET) ≜
     i1xi = "init"  ∧  it ∈ TT  ∧  i1xi′ = "active"
   ∧ t1xi = "idle"  ∧  t1xi′ = "active"  ∧  t1xtv′ = it
   ∧ ov = ["e0" ↦ iav, "e1" ↦ iv]
   ∧ ot = ["e0" ↦ it, "e1" ↦ it]
   ∧ last = {"e1"}
   ∧ unchanged(e6xq,e8xq);
  ...
END
```

Figure 10: Simple form of cTLA Process *Temperature Update*

4 Formalizing Collaborations

The concept of UML 2.0 collaborations as introduced in Sect. 2 is rather structural and as such *"describes a structure of collaborating elements"* [24]. Although UML enables collaborations, being so-called *behaviored classifiers*, refer to behaviors in form of interactions, state machines or activities, the coordination of these behaviors is not elaborated in detail. For our approach, however, in which we want to specify systems completely by composing collaborations, the behavioral part and the coordination of behavioral descriptions are essential. Therefore, we understand collaborations first and foremost as processes jointly executed by a set of participants. Composing systems from collaborations corresponds then to the task of synchronizing these processes. This de-

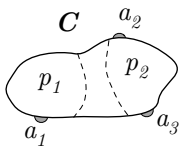


Figure 11: External view of a collaboration C

mands for a precise formal semantics describing both the behavior of collaborations as well as their composition. This does by far not exclude UML; on the contrary, such a well-defined formal basis enables us to use different UML diagrams, utilizing their specific advantages where appropriate. For this reason, we defined the cTLA style cTLA/c used to model collaborations in a way that several diagram types can be formalized.

To illustrate cTLA/c, Fig. 11 depicts a collaboration from an abstract, external viewpoint. It is a process between its participants p_1 and p_2. While most of the behavior may be executed internally to the collaboration, we need some mechanism to couple the collaborations with others during composition. For example, the end of one collaboration could trigger the start of another one, or collaborations may exchange data. For this, two principle solutions exist: communication by variables and synchronously executed actions. Only relying on the first one (i.e., allowing only producer/consumer synchronization) implies buffering, so that it would always take two execution steps for a collaboration to influence another one. In some cases, however, following the idea of constraint-oriented modeling (see Sect. 3), we may want to describe that events happen at the same time in several collaborations. Thus, both interaction principles can be useful, and cTLA/c is laid out to support both of them. For synchronous couplings, we simply conjoin the cTLA actions of different collaborations, while for buffered communication, we assume that the collaborations are linked to a special collaboration modeling the buffered communication. Both approaches use the cTLA coupling principle of joint actions [8, 10] (see also [21]). In Fig. 11, the externally visible cTLA actions a_1, a_2 and a_3 can be used to couple C_1 with other collaborations.

4.1 Elementary Collaborations

A collaboration that is not composed of other collaborations but describes its behavior directly by its actions and variables is called an *elementary collaboration*. For this, we use a simple cTLA process, as introduced in Sect. 2.1. Ignoring fairness assumptions and process parameters, a simple cTLA process can bee seen as a tuple $P_{simple} = \langle Act_{int}, Act_{ext}, Var, Init\rangle$, which declares the set of its variables, internal and external actions and an initial statement. To describe collaborations, we impose additional invariants on how cTLA processes are written. This basically defines the style of cTLA/c. For an elementary collaboration, we use the tuple

$$C_{el} = \langle Act_{int}, Act_{ext}, Init, Var_{loc}, Var_{com}, Part, p_{var}, p_{act}, NT\rangle.$$

Act_{int}, Act_{ext} and $Init$ have the same meaning as in P_{simple}. In addition, we define the participants of a collaboration by the set $Part = \{p_1 \dots p_n\}$. As these participants describe behavior to be executed by separate, physically distributed components, we assume only buffered communication between the different participants. This communication is expressed by special communication variables defined by the set Var_{com} while the state variables of the participants by Var_{loc} (in which $Var_{loc} \cap Var_{com} = \{\}$ holds and $Var_{loc} \cup Var_{loc}$ forms the set of all variables Var). Function $p_{var} \triangleq [Var \to Part]$ maps each variable to exactly one participant.

- A *local* variable v_l can be read and written only by the participant assigned to it via the function p_{var}. These variables are used to model local data, status of timers or the history of what has happened so far, to synchronize interactions with other participants.
- A *communication* variable v_c is a bag. It can be read by one participant only while the other participants may add elements. We attach v_c to the partition which can read it and constrain the function p_{var} accordingly.

The actions modeling interaction with the environment of the collaboration are described by the set act_{ext}, while act_{int} models behavior that is not visible from the outside. Similarly to the variables, each action is attached to a participant via function $p_{act} \triangleq [Act \to Part]$. An

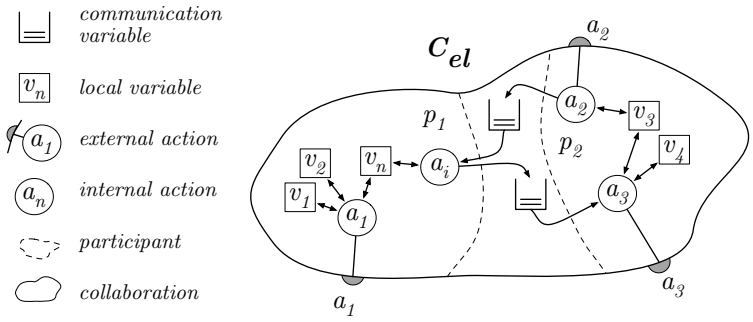


Figure 12: A cTLA/c process for an elementary collaboration C_{el}

action a_i attached to participant p_i may access local variables that are assigned to p_i as well. In addition, it may add elements to all communication variables of the neighboring collaborations, and receive from communication variables assigned to p_i. These definitions implied by cTLA/c on elementary collaborations are illustrated in Fig. 12. It is basically a bipartite graph showing the relationships between actions, participants and local and communication variables.

For the execution of components, we use state machines as expressed by cTLA/e, where actions correspond to state machine transitions that are triggered either by the expiration of a timer or the arrival of a signal. To enable an easier mapping from the actions of cTLA/c to the actions of cTLA/e in form of an implication $S_{cTLA/e} \implies S_{cTLA/c}$, we require also cTLA/c actions to be triggered. As the exact mode of triggering depends highly on the particular concepts of the diagrams formalized, we simply take the tuple element NT to mark all actions that are not triggered. Internal actions have always to be triggered such that NT may only include external actions.

4.2 Compositional Collaborations

A *compositional* collaboration refers to other collaborations and composes their behavior to describe its own behavior. For the description of a compositional collaboration, we use the tuple

$$C_{comp} = \langle Act_{int}, Act_{ext}, Part, p_{act}, Cu, bind, NT \rangle$$

As for the elementary collaboration, C_{comp} declares a set of internal and external actions, a set of participants, and a function p_{act} that assigns each action to one participant. In addition, there is the set Cu of collaboration uses. The participants of the instantiated collaborations are mapped to the participants of the collaboration under construction by function $bind \triangleq [Part \to Part]$. This enables us to use different partition names for the elementary and for the compositional collaborations which can be mapped into each other. Fig. 13 gives an illustration. For formal simplicity, there are no variables defined directly within a compositional collaboration. Coupling logic that needs variables and cannot be expressed by action couplings only, can be included by using dedicated collaborations.

As mentioned before, we use joined actions to couple collaborations. In cTLA/c there are some restrictions concerning the topology of the collaborations, and taking into account that internal actions must be triggered. Each action within a compositional collaboration is a conjunction of one action of each the n collaborations listed in Cu:

$$act = \bigwedge_{i=1\ldots n} a_i$$

with a_i being an external action or a stuttering step of collaboration use i. For the action coupling the following constraints must hold:

- Only actions mapped to the same participant may be coupled with each other. Collaborations not bound to the same collaboration role must stutter.
- Within each set of coupled actions, at most one action may be triggering.
- If none of a set of coupled actions has a link to the environment of the composite collaboration C_{comp}, the joint action act is internal. In this case, exactly one of the bound actions must be triggered.
- If one of a number of bound actions is linked to the environment of C_{comp}, the joint action is external. Here, either one or none of the joined actions must be triggered.

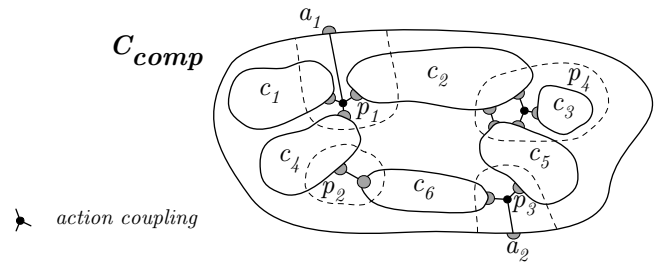


Figure 13: A cTLA/c process for a compositional collaboration C_{comp}

Following composition concept of cTLA, the compositional cTLA/c tuple C_{comp} can be expanded to an elementary collaboration such that hierarchical collaboration structures are possible.

We abstain from describing a special syntax for the constraints introduced by cTLA/c as we consider it only as a semantic concept behind other languages, such as UML activities, as described in the following. Once a specification semantically has the form of cTLA/c, it can be taken by our transformators and code generators to create an executable system.

5 Activities for Elementary Collaborations

In SPACE, we use UML 2.0 activities to express the behavior of collaborations as introduced in Sect. 2. A UML 2.0 collaboration is complemented by an activity which uses one separate activity partition for each collaboration role. In the terms of cTLA/c, an activity partition corresponds to a collaboration participant.

As already pointed out in Sect. 2, the semantics of UML 2.0 activities is based on Petri nets [24]. Thus, an activity basically describes a state transition system, with the token movements as the transitions and the placement of tokens within the graph as the states. In consequence, the variables of a cTLA/c specification model the actual token placement on the activity while its actions specify the flow of tokens between places. Activity edges may cross partition borders. According to the cTLA/c definition and due to the fact that the partitions are implemented by distributed components, flows moving between partitions are modeled by means of communication buffers while places assigned to activity nodes are represented in cTLA/c by local variables.

We discussed above that certain variables of a cTLA/c collaboration may be triggers. For activities, triggers are represented by initial nodes starting a flow in the beginning, timer nodes which trigger a flow upon expiration, and edges crossing a partition border in which a token in the corresponding communication buffer is triggered to forward in the receiving partition[7]. Moreover, a token may be triggered from the activity environment which is expressed by flows passing pins at the border of call behavior actions. This leads to flows which — from a local view of a single activity — are non-triggered. Of course, in order to achieve lively flows, non-triggered partial flows have to be connected with other flows in a way that in the system description all flows start at a triggering node.

The properties on the triggering of flows lead to constraints on the alignment of places to activity nodes since not every flow could be triggered if a token can rest at any node. In general, we allow places only on nodes modeling either entities that can trigger or locations in which a token has to wait for another flow to synchronize. The first group comprises initial nodes, timer nodes, and crossing points of edges through partition borders while the second one covers the following cases:

- a join node in which a token must wait if the other incoming edges are not yet filled,
- a waiting decision node (see Sect. 2) enabling a token to leave via different joins (see [19]), where are token has to wait if none of the succeeding joins can fire.

In contrast, tokens do not rest within call operation actions. This is not useful as no trigger is available which may lead a flow to leave the call operation action after the invoked operation is finished (see [20]). Instead, we consider the execution of the operation "on-the-fly" by a token passing its call operation action.

[7]This definition results from the need to transform activities to the special UML state machines forming the input of our code generators (see [19]).

In the following, an activity is given by the tuple

$$A \triangleq \langle nodes, edges, type, part, location, guard \rangle$$

with *nodes* as the set of activity nodes. $edges \subseteq \text{SUBSET } nodes \times nodes$[8] describes the set of activity edges, while the function $type \in [nodes \rightarrow Type]$ assigns to each activity node the node type which is an element of the set $Type = \{$*initial*, *fork*, *join*, *merge*, *decision*, *waiting*, *timer*, *receive*, *send*, *input*, *output*, *operation*, *callBehavior*, *inputPin*, *outputPin*$\}$. *part* models the set of partitions, while $location \triangleq [nodes \cup edges \rightarrow \text{SUBSET } part \backslash \{\}]$ assigns each node and edge a non-empty set of partitions. Here, all nodes except for call behavior actions must only be mapped to exactly one partition while edges may belong to several partitions as they can arbitrarily cross partition borders. Guards are assigned to all edges following a decision node by function $guard \triangleq [edges \rightarrow Guards \cup \{\}]$. In addition to the tuple A, we define functions *outgoing* and *incoming* as $\triangleq [node \rightarrow \text{SUBSET } edges]$ that give us the set of incoming and outgoing edges of a node. In particular, only decision and fork nodes have more than one outgoing edge, and only merge and join nodes have more than one incoming edge.

To define the semantics of activities using cTLA/c, we opted for an approach that makes directly use of the composition mechanisms of cTLA.

1. We describe for some node types[9] of an activity a separate cTLA process which are introduced in Sect. 5.1. This already helps to understand the semantics of the nodes.

2. To obtain a cTLA/c representation C_A for an activity A, we define C_A as a compositional cTLA process and include for every activity node an instance of the corresponding cTLA process modeling the node.

3. Thereafter, we create the actions of C_A specifying A's flows of tokens. In particular, we traverse the edges of the activity. At a starting point of a flow, a new cTLA action is created which is amended successively when the traversal passes an activity node. The creation and amendment of the actions is guided by a set of production rules introduced in Sect. 5.2. In Sect. 5.3 we clarify the action creation with an example.

4. As C_A is yet a compositional cTLA specification, we finally expand it to an equivalent simple process as discussed in Sect. 3.2. The result of this transformation is the formal model of the activity.

5.1 cTLA Processes for Activity Elements

As mentioned above, we model flows passing partition borders by buffered queues. In consequence, the communication variables used for the communication between the participants in the cTLA/c definition are described by buffers storing tokens. Thus, in the tuple C_{el}, introduced in Sect. 4, we define the element Var_{com} as the set of queues of tokens containing a separate element for each crossing of an edge through a partition border. Each queue is located at the partition entered by the edge and the tuple element p_{var} is accordingly defined. The places on the activity nodes on which tokens may rest can store at most one token each. For input nodes, timers and waiting decisions, we assign one place for the overall node while joins are provided with a separate place for each incoming edge not leaving a waiting decision. In our cTLA/c processes, every place is described by a boolean flag each modeling if the place is filled by a token or not. Moreover, UML activities enable to use auxiliary variables to express data. As tokens can store data in local signatures[10], we further need variables storing their current signature if they rest on a place. In consequence, we define the set Var_{loc} of local states as the union of the flags describing the places and the cTLA representations of the auxiliary variables resp. token signature stores. Each place and the assigned signature store are directly linked with an activity node local to a particular partition. In addition, we assume that each auxiliary variable is also local to a partition, so that we can define the mapping p_{var} in a straightforward manner.

The cTLA process types modeling the different activity node types have to fulfill these constraints. Thus, initial nodes, joins (including waiting decisions), timers and receive nodes are represented by cTLA

[8] SUBSET S is also called power set of S

[9] For merges, forks, and final nodes special cTLA processes are not necessary as we will discuss later.

[10] This implies UML object flows [24], which we do not consider here in detail.

processes defining their places and token signature stores. The cTLA process modeling edges crossing partition borders defines meanwhile the communication queue specifying the partition change. In addition, for each partition, we describe a cTLA process storing the local auxiliary variables. Decision, sending and operation nodes are represented as stateless cTLA processes since that helps to encapsulate their specific properties. For brevity, we introduce only the cTLA processes for initial nodes, decisions, timers, operations as well as for transfer edges which are necessary to understand the example sketched in Sect. 5.3. A complete documentation comprising the cTLA processes for all activity nodes is provided in [18].

Before discussing the cTLA processes in detail, we introduce some generic data types used as process parameters. *VT* describes the types of all auxiliary variables in a partition. Here, we assume that a list of variables is expressed by a single record element. The signature set of the tokens is represented by the type *TT*, while *ET* is an enumeration providing each edge in an activity a unique identifier.

Initial Nodes The variable i is the flag describing the place at an initial node. The place is only filled in the initial system state (value *"idle"*) while it will remain empty when the activity is running (value *"active"*). The leaving of the token from the initial node is modeled by the action *start* which must only be executed if the token is in its place (i.e., i = *"idle"*) and removes the token (i' = *"active"*). The action parameter t specifies the signature of the token. Since that is not defined explicitly, it may contain any correct value of set *TT*. *start* is a trigger action modeling the start of a new token flow.

```
PROCESS Initial(TT: Any)
VARIABLES i: {"init","active"};
INIT ≜ i = "init";
ACTIONS
  start(t: TT) ≜ i = "init" ∧ t ∈ TT ∧ i′ = "active";
END
```

Timer Nodes A timer node[11] contains also a place on which a token may rest. In the corresponding cTLA process that was already introduced in Fig. 8, we use a boolean flag i and a store *tv* for the token signature. An idle timer is activated by an arriving token, represented by the cTLA action *start*. This action uses the parameter *it* to model the parameters of the arriving token. It is enabled if the place is empty (i.e., i = *"idle"*) which will, consequently, being filled with the token (i.e., *i'* = *"active"* ∧ *tv'* = *it*). As we do not model time explicitly yet, the timer can expire at any time, described by the action *expire* which can only be executed if the place is filled. Here, i is set to *"idle"* and the parameter *ot* specifying the signature of the leaving token is assigned with *tv*. The third action, *expireAndRestart* models that the timer expires but is restarted within the same step. This extra action is needed, as a conjunction of action *start* and *expire* would assign contradicting values to the state which would block it forever. *expire* and *expireAndRestart* are trigger actions.

[11] Technically, a timer node is a UML accept event action with a time event as its trigger.

Transfer Edges The queue modeling the transfer of a token from one partition to another one is modeled by the variable q. It stores for every received token the corresponding signature and delivers this information in FIFO order. The arrival of a token with the signature *it* at the partition border is specified by the action *send* while *receive* models the consumption of a token with signature *ot*. According to this definition, the action *start* is assigned to the partition from which the edge leads to the partition border while *receive* is part of the one consuming the token. *receive* is a trigger action.

```
PROCESS Transfer(TT : Any)
VARIABLES
  q: Queue(TT);
INIT ≜ q = EMPTY;
ACTIONS
  send(it: TT) ≜ q′ = append(q,it);
  receive(ot: TT) ≜ q ≠ EMPTY ∧ ot = first(q) ∧ q′ = tail(q);
END
```

Call Operation Actions An operation may change the values of local auxiliary variables of the partition, in which it is defined, as well as the signature of the token flowing through it. We describe operations by the stateless cTLA process *Operation*, which takes two functions as parameters *nv* and *nt*. These parameters reflect that a call operation

action may change both the signature of the tokens and the auxiliary variables. Consequently, *nv* is a function that describes the operation's effect on the values of the auxiliary variables. Similarly, *nt* describes the deriving of new token values. The method *execute* models the computation of new values according to these functions. As action parameters it uses *iv* expressing the auxiliary variable setting and *it* specifying the token signature before executing the operation. The new value of the auxiliary variables and the new token signature are described by the action parameters *ov* resp. *ot*.

```
PROCESS Operation(nv: [VT × TT → VT]; nt: [VT × TT → TT])
ACTIONS
  execute (iv: VT; it: TT; ov: VT; ot: TT) ≜
     ov = nv[iv,it] ∧ ot = no[iv,it];
END
```

Decision Nodes A decision is specified by a stateless cTLA process, too. It may have n outgoing edges modeled by the process parameter of the same name. The other parameter is a function characterizing the guards of the outgoing edges. Its domain set is a tuple defining the identifier of the guard as a number between 1 and n as well as the current value of the auxiliary variables and the token signature. The tuples are mapped to boolean values. The action *decide* reflects a semantics according to which exactly one guard of a decision node has to be true. The parameter e refers to the number of the checked guard and the action may only be executed for this guard if all guards with smaller numbers *ed* are not executable and either the guard of e holds or e is the highest number. The latter condition reflects that one guard should always contain the value *else*.

```
PROCESS Decision (n: NATURALS; gu: [1..n × VT × TT] → BOOLEAN])
ACTIONS
  decide(e: 1..n; av: VT; t: TT) ≜
    ∀ ed ∈ {1..n}:
      ed < e ⇒ ¬ gu[ed,av,t] ∧ e = n ∨ gu[e,av,t];
END
```

5.2 Production Rules for cTLA/c Actions

The processes for the activity nodes explained in the last section are instantiated as part of the cTLA process for the activity C_A and constitute the state space for this process. They also declare actions for their respective nodes, which, in the following, have to be coupled in accordance with the activity edges building the system edges of C_A.

We decided to present the way producing the system actions from the local process actions as a set of rules, so that each activity element can be discussed separately. There are two types of rules:

- Rules that *create* a new action. These rules treat triggering nodes like timers or incoming transfer edges. As well as edges starting at an input or output pin of a call behavior action. They simply start the construction of a new action in C_A.
- Rules that *replace* an existing action. These rules model the continuation of a flow. They start at an edge that is not triggering, take the already produced action *act* for the upstream graph and add a conjunct c to the existing action, so that a new action $act^\star = act \wedge c$ is created. This new action replaces the existing one. Except for the special case in which a flow reaches the activity node that triggered it, this replacement corresponds to a cTLA process composition. The existing action is encapsulated as external action within a process and then composed in a compositional process together with another process encapsulating the additional sub-action c. The result can then be expanded to a simple cTLA process, which is equivalent to a (maybe more intuitive) replacement of the action. If a flow reaches the node from which it started, we have to replace the action specifying the triggering by another one modeling both the triggering and the consumption of a token. E.g., for a timer, the action *expire* defined in process *Timer* (see Fig. 8) has to be exchanged by *expireAndRestart*. In this case, we have a genuine replacement.

Each production rule is presented in two parts. The first compartment collects the preconditions of the rule. It refers to the structure of an activity and defines the activity edges resp. nodes for which the rule can be used. Moreover, the cTLA action to be replaced is listed. As an additional precondition, we need to remember when traversing an activity which of its edges have still to be visited. In a production rule, we therefore use the function $toVisit \in [Act \rightarrow \text{SUBSET } ET]$ storing for a particular cTLA action the edges still to be passed.

The second compartment shows the effect of the rule. It gives instructions whether a new action should be created or an existing one should be replaced, and how the emerging action is constructed. It also declares any changes to the function $toVisit$ by updating the set of edges still to be visited for an action.

The construction of an action begins with one of the starting points of an activity, that means at initial nodes, at the exit of timers (which means expiration), when an edge enters a partition, or when an external signal arrives. The rules INITIAL, TIMEREXPIRE, TRANSFERENTER and RECEIVE introduced below describe hereby how the action is written. Afterwards, other rules are applied to it guided by the nodes and edges that follow in the activity graph. A new action is created by adding conjuncts to the original one. In case we reach a decision or join node, the action created from the incoming graph is replaced by an entire set of actions. The construction of an action is finished when a new stable state is reached in the activity partition, that means that we either leave the partition, rest in a join or waiting decision, set a timer or reach a final or receiving node. Moreover, the leaving of a flow through the pin of a call behavior action is also a stopping point.

The actions under construction have the signature

```
act(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
    is, os: SUBSET TT; last: SUBSET ET)
```

The parameter *it* specifies the value of the token when the flow starts. While the function *ot* describes the token signature after leaving a particular edge. Similarly, parameters *iv* and *ov* describe the values of the local variables after the flow starts and after traversing a particular edge. Signals sent within an action are described with parameter *os*, and signals received by *is*. Parameter *last* keeps track of the edges in an action after those the flow stops. This is needed to support the storage of the auxiliary variables discussed in Sect. 5.1.

In the following we will show the rules for initial nodes, merges, forks, timers, operations, transfer edges as well as decisions and flow final nodes. The remaining rules are listed in [18].

Initial Nodes As an initial node is a trigger, it is the startpoint for the production of an action. The rule is enabled for an initial node i with an outgoing edge e. It creates action *act*, which is coupled with action *start* to the process instance corresponding to the initial node, p_i. As the flow is not yet finished, *last* is empty. The node neither produces any output signals ($os = \{\}$) and does not change the value of the variables or token, so that *ov* and *ot* remember their respective initial values for this edge. As we continue the production of the action with whatever comes after edge e, we store it as still to be visited.

INITIAL

$\exists\, i,e$: $type(i)$= *"initial"*
$outgoing(i)$=$\{e\}$

→ *Create* act *with*

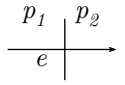
```
act(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
                is, os: SUBSET TT; last: SUBSET ET) ≜
    p_i.start(it)
  ∧ ov = [ e ↦ iv ]  ∧  ot = [ e ↦ it ]
  ∧ os = {}  ∧  last = {}
```

$toVisit'$=[$toVisit$ EXCEPT ![$act^\star$]=$\{e\}$]

Transfer Edges Edges crossing partition borders are handled by two rules, TRANSFERLEAVE modeling the leaving of the current partition, and TRANSFERENTER for edges entering a partition. TRANSFERLEAVE is a rule that adds a conjunct invoking *send* on process p_t modeling the buffered communication. As the flow ends where it leaves the partition, the edge is removed from the edges that must be visited but entered to the set *last* describing final edges. The rule for receiving (not shown) is similar to the expiration of a timer or an initial node. It creates a new action referring to the triggered action *receive* of the transfer process.

TRANSFERLEAVE

$\exists\, e,p_1,p_2,act$: $e \in edges$
$location(e)$=$\langle p_1,p_2\rangle$
$e \in toVisit[act]$



→ *Replace* act *by* act* *with*

```
act*(...) ≜ ∃ last_o: act(it, ot, iv, ov, is, os, last_o)
  ∧ p_t.send(ot[e])  ∧  last = last_o ∪ {e}
```

$toVisit'$=[$toVisit$ EXCEPT ![$act^\star$]=$toVisit[act]\backslash\{e\}$]

Flow Final Nodes Flow final nodes simply terminate a token flow. The original action is finished by noting its last edge.

FlowFinal

$\exists\, z,e,act$: $type(z)=$ *"flowFinal"*
$incoming(z)=\{e\}$
$e \in toVisit[act]$

$\longrightarrow$ *Replace* `act` *by* $\texttt{act}^\star$ *with*

$\texttt{act}^\star(\ldots) \triangleq \exists\ \texttt{last}_\texttt{o}\texttt{: act(it, ot, iv, ov, is, os, last}_\texttt{o}\texttt{)}$
$\quad \wedge\ \texttt{last} = \texttt{last} \cup \{\texttt{e}\}$

$toVisit'=[toVisit$ EXCEPT $![act^\star]=toVisit[act]\backslash\{e\}]$

Call Operation Actions As described above, operations are modeled as functions that assign new values to the token passing through it as well as the variables, modeled by the cTLA process *Operation* with its action *execute*. This action is coupled with the original one, and the production continues with the outgoing edge of the operation. The values *ot* and *ov* for the outgoing edge j reflect the changes carried out by the operation which are described by the action parameters new_v and new_t.

Operation

$\exists\, op,e,j,act$: $type(op)=$ *"operation"*
$incoming(op)=\{e\}$
$outgoing(op)=\{j\}$
$e \in toVisit[act]$

$\longrightarrow$ *Replace* `act` *by* $\texttt{act}^\star$ *with*

$\texttt{act}^\star(\ldots) \triangleq \exists\ \texttt{ot}_\texttt{o}\texttt{, ov}_\texttt{o}\texttt{, new}_\texttt{v}\texttt{, new}_\texttt{t}\texttt{:}$
$\quad \texttt{act(it, ot}_\texttt{o}\texttt{, iv, ov, is, os, last)}$
$\quad \wedge\ \texttt{p}_\texttt{o}\texttt{.execute(iv}_\texttt{o}\texttt{[e],ot}_\texttt{o}\texttt{[e],new}_\texttt{v}\texttt{,new}_\texttt{t}\texttt{)}$
$\quad \wedge\ \texttt{ov} = \texttt{[ov}_\texttt{o}\ \texttt{EXCEPT !j} \mapsto \texttt{new}_\texttt{v}\texttt{]}$
$\quad \wedge\ \texttt{ot} = \texttt{[ot}_\texttt{o}\ \texttt{EXCEPT !j} \mapsto \texttt{new}_\texttt{t}\texttt{]}$

$toVisit'=[toVisit$ EXCEPT $![act^\star]=toVisit[act]\cup\{j\}\backslash\{e\}]$

Timer For a timer, three rules determine the creation and coupling of actions. The expiration of a timer triggers an action. Rule TimerExpire defines therefore the creation of a new action that starts with the outgoing edge of the timer, similarly to an initial node. It couples the *expire* action of the timer process p_t, so that this action is only enabled if the timer is active.

TimerExpire

$\exists\, t,e$: $type(t)=$ *"timer"*
$outgoing(f)=\{e\}$

$\longrightarrow$ *Create* `act` *with*

$\texttt{act(}\ldots\texttt{)} \triangleq$
$\quad \texttt{p}_\texttt{t}\texttt{.expire(it)}$
$\quad \wedge\ \texttt{ov} = \texttt{[e} \mapsto \texttt{iv]}\ \wedge\ \texttt{ot} = \texttt{[e} \mapsto \texttt{it]}$
$\quad \wedge\ \texttt{os} = \texttt{\{\}}\ \wedge\ \texttt{last} = \texttt{\{\}}$

$toVisit'=[toVisit$ EXCEPT $![act^\star]=\{e\}]$

To start a timer, action $p_t.start()$ is conjoined with the action modeling the rest of the subgraph. It must only be used if the flow started from another element than the timer itself. In the precondition, this is stated by adding the condition $t \notin visited[act]$ describing that the timer t is not in the list of visited nodes.

TimerStart

$\exists\, t,e,act$: $type(t)=$ *"timer"*
$incoming(f)=\{e\}$
$e \in toVisited[act]$
$t \notin visited[act]$

$\longrightarrow$ *Replace* `act` *by* $\texttt{act}^\star$ *with*

$\texttt{act}^\star(\ldots) \triangleq$
$\quad \exists\ \texttt{last}_\texttt{o}\texttt{: act(it, ot, iv, ov, is, os, last}_\texttt{o}\texttt{)}$
$\quad\quad \wedge\ \texttt{p}_\texttt{t}\texttt{.start(ot[e])}$
$\quad\quad \wedge\ \texttt{last}' = \texttt{last} \cup \texttt{\{e\}}$

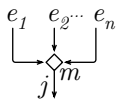
$toVisit'=[toVisit$ EXCEPT $![act^\star]=toVisit[act]\backslash\{e\}]$

Rule TimerExpireRestart (not shown) is used instead of TimerStart if a timer expires and is restarted within the same action. This is stated in the precondition by $t \in visited[act]$. For the result, the only difference is that action *expireAndRestart* instead of just *expire* or *start* is called.

Merge Nodes Merge nodes copy the behavior following the node to the behavior started before the node. The rule is applied to all actions already produced for each of the incoming edges. As an merge does neither change the token signature nor the values of the auxiliary variables, *ot* and *ov* are set to the same values for j as for the incoming edge e_p.

MERGE

$\exists\, m,e_p,j,act$: $type(m)=$ "merge"
$e_p \in incoming(m)$
$outgoing(m)=\{j\}$
$e_p \in toVisit[act]$



→ *Replace* `act` *by* `act`* *with*

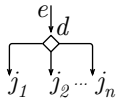
```
act*(...) ≙
  ∃ ov_o, ot_o: act(it, ot_o, iv, ov_o, is, os, last)
   ∧ ov = [ov_o EXCEPT !j ↦ ov_o[e_p] ]
   ∧ ot = [ot_o EXCEPT !j ↦ ot_o[e_p] ]
```

$toVisit'=[toVisit$ EXCEPT $![act^\star]=toVisit[act]\cup\{j\}\setminus\{e_p\}]$

Decision Nodes Decision nodes multiply the incoming actions by the number of its outgoing edges (the alternatives). Therefore, the incoming action is replaced by a set of actions. The original content of the incoming action is maintained, it is just expanded with the additional call of the decision action of the decision process.

DECISION

$\exists\, d,e,act$: $type(d)=$ "decision"
$incoming(d)=\{e\}$
$outgoing(d)=\{i_1 \ldots j_n\}$
$e \in toVisit[act]$



→ *Replace* `act` *by* `act`$_p$ *for all* $p \in \{1 \ldots n\}$ *with*

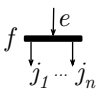
```
act_p(...) ≙ ∃ ov_o, ot_o: act(it, ot_o, iv, ov_o, is, os, last)
    ∧ p_d.decide(p, ov[e], ot[e])
    ∧ ov = [ov_o EXCEPT !j_p↦ ov_o[e] ]
    ∧ ot = [ot_o EXCEPT !j_p↦ ot_o[e] ]
```

$toVisit'=[toVisit$ EXCEPT $![act_p]=toVisit[act]\cup\{j_p\}\setminus\{e\}]$

Fork Nodes Forks multiply a token and emit one token on each outgoing edge. All the behaviors implied until all tokens rest, are executed within one step, so that the whole behavior has to be modeled within one action. Therefore, all outgoing edges are added to the edges still to be visited for the action under construction. Since the fork does not change the value of the tokens or variables, the functions for token and variable values are updated to match the incoming values for each outgoing edge.

FORK

$\exists\, f,e,\{j_1 \ldots j_n\},act$: $type(f)=$ "fork"
$incoming(f)=\{e\}$
$outgoing(f)=\{j_1 \ldots j_n\}$
$e \in toVisit[act]$



→ *Replace* `act` *by* `act`* *with*

```
act*(...) ≙ ∃ ov_o, ot_o: act(it, ot, iv, ov, is, os,last)
   ∧ ov = [ov_o EXCEPT !j_1 ↦ ov_o[e] EXCEPT ! ···
                EXCEPT !j_n ↦ ov_o[e] ]
   ∧ ot = [ot_o EXCEPT !j_1 ↦ ot_o[e] EXCEPT ! ···
                EXCEPT !j_n ↦ ot_o[e] ]
```

$toVisit'=[toVisit$ EXCEPT $![act^\star]=toVisit[act]\cup\{j_1,\ldots,j_n\}\setminus\{e\}]$

The produced action couplings conform to the constraints in cTLA/c.

- The production of an action always stays within the partition where the production started. Edges leaving a partition terminate the production of an action by a corresponding send action of a transfer process. Consequently, all produced actions can be assigned to exactly one participant of the cTLA/c process under construction.
- Actions are by default internal (i.e., $\in Act_{int}$). Only if they pass an input or output node (such as *update display* or *update device* in Fig. 4), they are declared external.
- Just for flows starting at an input or output node, actions are created that do not contain a trigger. According to the definition above, however, these actions are external and the cTLA/c claims for non-triggered actions are met. These actions are added to the set NT listing the non-triggered actions. Due to the structure of

activities and the layout of the rules, a sub-graph corresponding to an action can never contain more than one trigger[12].

5.3 Example

We will now use the rules to produce parts of the cTLA/c process for the activity *Temperature Update* given in Fig. 4. First, we instantiate processes *i1*, *t1*, *o1*, *d1*, *e6*, *e8* and *o2* for the corresponding activity nodes, as shown in the corresponding cTLA process in Fig. 9. In the following, we will stepwise create some of the actions for the partition of the heater.

Step 1: We choose to start with the initial node i_1 and apply rule INITIAL to edge e_0, which leads to the construction of action act_1. As no signal has yet been sent, *os* is empty. There is no final edge, as the flow continues. The variables did not change with the initial node, such that *ov* notes the original value *iv* for edge e_0. The same applied for the value of the token managed by *ot*.

```
act1(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
     is, os: SUBSET TT; last: SUBSET ET) ≜
     i1.start(it)
  ∧ ov = ["e0"↦ iv] ∧ ot = ["e0"↦ it]
  ∧ os = {} ∧ last = {}
```

Step 2: We continue with this flow by applying rule MERGE to edge e_1 and the already created action act_1. It is replaced by act_2, which is an extension of act_1. A merge does not change tokens or variables, so *ov* and *ot* are complemented with an entry for edge e_1.

```
act2(...) ≜
∃ ov_o, ot_o: act1(it, ov_o, iv, ot_o, os, is, last)
   ∧ ov = [ov_o EXCEPT !"e1"↦ ov_o("e0")]
   ∧ ot = [ot_o EXCEPT !"e1"↦ ot_o("e0")]
```

We expand act_2 by replacing act_1 with its actual definition:

[12]In faulty activities, sub-graphs may exist that have neither a triggering element nor a connection via a parameter node. These constructs would not cause an action to be produced, as none of the actions could be applied in the first place. These situations may be detected by syntactic inspections. As such sub-graphs do not express any useful behavior, they are forbidden.

```
act2(...) ≜
∃ ov_o, ot_o:
     i1.start(it)
   ∧ ov_o = ["e0"↦ iv] ∧ ot_o = ["e0"↦ it]
   ∧ os = {} ∧ last = {}
 ∧ ov = [[ "e0"↦ iv] EXCEPT ! "e1"↦ ["e0"↦ iv]("e0")]
 ∧ ot = [[ "e0"↦ it] EXCEPT ! "e1"↦ ["e0"↦ it]("e0")]
```

We can replace the existentially quantified terms ov_o and ot_o by the equal function definitions. Further, $[\text{"}e_0\text{"} \mapsto x](\text{"}e_0\text{"})$ is of course x, so that we can simplify act_2:

```
act2(...) ≜
    i1.start(it)
  ∧ os = {} ∧ last = {}
  ∧ ov = ["e0"↦ iv, "e1"↦ iv]
  ∧ ot = ["e0"↦ it, "e1"↦ it]
```

Step 3: Rule TIMERSTART is now applicable to act_2 and edge e_1. It extends act_2 by adding action $t1.start$ from the timer process and updates last.

```
act3(...) ≜
  ∃ last_o: act2(it, ot, iv, ov, is, os, last_o)
   ∧ t1.start(ot["e1"])
   ∧ last = last_o ∪ {"e1"}
```

After expansion of act_2 and removal of true conjuncts, we get

```
act3(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
     is, os: SUBSET TT; last: SUBSET ET) ≜
     i1.start(it)
  ∧ os = {}
  ∧ ov = ["e0"↦ iv, "e1"↦ iv]
  ∧ ot = ["e0"↦ it, "e1"↦ it]
  ∧ t1.start(it)
  ∧ last = {"e1"}
```

No more rules can be applied to act_3, as there are no more edges to visit for this action. The action is complete now and can be added to the cTLA process describing *Temperature Update* (see Fig. 9).

Step 4: Rule TIMEREXPIRE can be applied to edge e_2, which results in the creation of act_4:

```
act4(...) ≜
    t1.expire(it)
  ∧ ov = ["e2"↦ iv]
  ∧ ot = ["e2"↦ it]
  ∧ os = {} ∧ last = {}
```

Step 5: Edge e_2 flows into fork f_1, so that rule FORK may be applied to act_4. It replaces act_4 by act_5 (here with act_4 already expanded).

```
act5(...) ≜
    t1.expire(it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv ]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it ]
  ∧ os = {} ∧ last = {}
```

$toVisit[act_5]$ contains now both outgoing edges, e_3 and e_4.

Step 6: Following edge e_3 into m_1, rule MERGE is applicable. It simply complements *ov* and *ot* with entries for edge e_1.

```
act6(...) ≜
    t1.expire(it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv ]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it ]
  ∧ os = {} ∧ last = {}
```

Step 7: Continuing edge e_1 we enter timer t_1. As the currently traversed activity flow started at this node, we apply rule TIMERSETEXPIRE instead of rule TIMERSET. Therefore, the action *expire* of process *t1* is replaced by *expireAndRestart* that handles a flow immediately restarting the timer from which is was triggered.

```
act7(...) ≜
    t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv ]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it ]
  ∧ os = {} ∧ last = {"e1"}
```

Step 8: To handle the operation by following edge e_4, we replace act_7 with act_8 that adds the conjuncts according to rule OPERATION. The operation is a function *o1nav* that computes new values for all the variables in the partition, and an *o1nto* that computes the value of a new token.

```
act8(...) ≜
    t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it] ]
  ∧ os = {} ∧ last = {"e1"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
```

Step 9: We apply rule DECISION. It replaces action act_8 with one action for each outgoing branch. For the branch of edge e_6 we get act_9, for the e_7 branch we get act_{10}

```
act9(...) ≜
    t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
           "e5"↦ o1nav[iv,it], "e6"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e6"↦ o1nto[iv,it]] ]
  ∧ os = {} ∧ last = {"e1"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
  ∧ d1.decide(1,o1nav[iv,it], o1nto[iv,it])

act10(...) ≜
    t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e7"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e7"↦ o1nto[iv,it]] ]
  ∧ os = {} ∧ last = {"e1"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
  ∧ d1.decide(2,o1nav[iv,it], o1nto[iv,it])
```

Step 10: When we continue with action act_9 and edge e_6, we apply rule TRANSFERSEND and replace it by action act_{11}, that simply adds an conjunction sending the token. As the set of edges to visit is empty for this edge, this is an action present in the final process.

```
act11(...) ≜
    t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e6"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e6"↦ o1nto[iv,it] ]
  ∧ os = {} ∧ last = {"e1", "e6"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
```

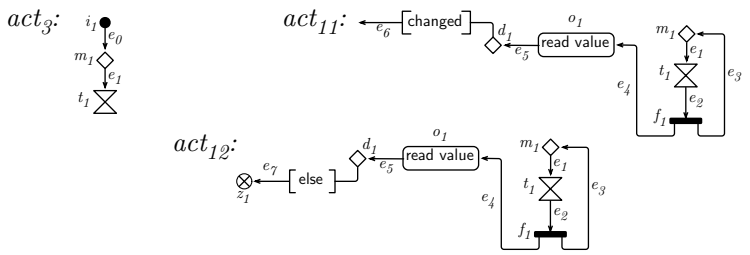


Figure 14: The subgraphs covered by the produced actions

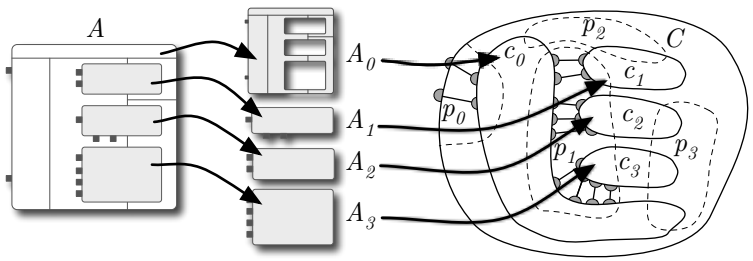


Figure 15: Mapping activities to a compositional cTLA/c process C

```
  ∧ d1.decide(1,o1nav[iv,it], o1nto[iv,it])
  ∧ e6.send(o1nto[iv,it])
```

Step 11 Also $act10$ may be finalized by applying rule FLOWFINAL with edge e_7. We obtain action act_{12}, which is like act_3 and act_{11} one of the final coupling actions.

```
act12(...) ≜
     t1.expireAndRestart(it, it) this has input and output
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e7"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e7"↦ o1nto[iv,it]] ]
  ∧ os = {} ∧ last = {"e1", "e7"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
  ∧ d1.decide(2,o1nav[iv,it], o1nto[iv,it])
```

Above, we sketched the production of the three actions act_3, act_{11} and act_{12} modeling the flows listed in Fig. 14. In a similar way, we can produce the other cTLA actions modeling flows in the activities of our *Mobile Home Control* example.

All-in-all, the production rules give a powerful means to formalize UML 2.0 activities as they are used in SPACE. If necessary, the generation process can be automated as done in [29] for checking activities with the model checker TLC[13] [32].

[13] As TLC is based on the TLA modeling language TLA$^+$, Slåtten had to modify the cTLA descriptions which, due to the foundation of cTLA (see Sect. 3), was straightforward.

6 Composing Collaborations by Activities

Following the style of cTLA/c, a composite activity A referring to n sub-activities $A_1, \ldots, A_n$ is modeled by a compositional cTLA/c tuple C_{comp} as described in Sect. 4.2. For each collaboration use (and consequently call behavior action of A) that is declared in the UML specification, $C_{comp}.Cu$ contains the elementary cTLA/C tuples $c_{el_1}, \ldots, c_{el_n}$. The compositional cTLA process C realizing C_{comp} includes the cTLA process instances $c_1, \ldots, c_n$ specifying the elementary collaborations. Besides the behavior within the call behavior actions, there may be arbitrary complex logic in A, coupling the referred sub-activities. This behavior of A without its sub-activities is an activity itself represented in C as its own cTLA/c process c_0. Thus, if A contains n call behavior actions, the composite collaboration $C_{comp}.Cu$ has $n+1$ processes as elements, $c_0, \ldots, c_n$. As an example, we refer to the activity *Zone Session* depicted in Fig. 6. The mapping from the activity to a compositional cTLA/c process is illustrated in Fig 15. Activity A is cut into the activities modeled by the call behavior actions $g \equiv A_1$, $a \equiv A_2$, and $u \equiv A_3$ as well as the one surrounding the call behavior actions A_0. These are expressed by simple cTLA/c processes.

The participants $C_{comp}.Part$ of the compositional cTLA/c process correspond to the activity partitions resp. collaboration roles of the UML collaboration. Following the collaboration role binding of the UML collaboration resp. the topology and partition mapping of the corresponding activity, the function $C_{comp}.bind$ maps each external action of $c_0, \ldots, c_n$ to a participant in $C_{comp}.Part$.

6.1 Synchronous Coupling

The link between an activity (like A_0) and an activity referred from it (like A_1) is described by the input and output pins of call behavior actions. An input pin models a flow of tokens from A_0 to another activity A_i while an output pin specifies an opposite flow. Every pin has exactly one incoming as well as one outgoing edge which makes the formal definition of the coupling straightforward. Due to the production rules introduced in Sect. 5.2, an edge heading to a pin can be modeled by an arbitrary number of cTLA actions $outp_1, \ldots, outp_k$ while an edge leaving a pin is modeled by a number of cTLA actions $inp_1, \ldots, inp_l$. These actions use the action parameter signature introduced in Sect. 5.2. Assuming that an input or output pin of the call behavior action hosting activity A_i is reached by an edge with the marking e_o and left by the edge e_i, we define $k \cdot l$-many system actions $act_{q,r}$ with $q \in \{1..k\}, r \in \{1..l\}$ of the corresponding cTLA process c as follows[14] ($v, w \in \{0, i\}$, $v \neq w$):

$$
\begin{array}{l}
\mathtt{C.act_{q,r}(it:\ TT;\ ot:\ [ET \to TT];\ iv:\ VT;\ ov:\ [ET \to VT];} \\
\qquad \mathtt{is,\ os:\ SUBSET\ TT;\ last:\ SUBSET\ ET)} \triangleq \\
\exists\ \mathtt{ot_o,\ ov_o,\ is_o,\ os_o,\ last_o,\ ot_i,\ ov_i,\ is_i,\ os_i,\ last_i}: \\
\quad \mathtt{C_v.outp_q(it,ot_o,iv,ov_o,is_o,os_o,last_o)} \\
\wedge\ \mathtt{C_w.inp_r(ot_o["e_o"],ot_i,ov_o["e_o"],ov_i,is_i,os_i,last_i)} \\
\wedge\ \mathtt{ot = FMERGE(ot_o,ot_i)}\ \wedge\ \mathtt{ov = FMERGE(ov_o,ov_i)} \\
\wedge\ \mathtt{is = is_o \cup is_i}\ \wedge\ \mathtt{os = os_o \cup os_i} \\
\wedge\ \mathtt{last = (last_o \cup last_i) \setminus \{"e_o"\};}
\end{array}
$$

Given that the activities are syntactically correct and consistent with the UML collaborations they complement, these couplings produce valid cTLA/c process couplings.

- Due to the fact, that the pins are unambiguously allocated to one partition p, all output and input actions belong to p as well. Thus, we can assign the system action $c.act_{q,r}$ to p as well which will be reflected in the function p_{act} of tuple C_{comp}.
- In addition, $c.act_{q,r}$ contains a trigger if and only if $c_v.outp_q$ contains a trigger, too (i.e., $c.act_{q,r} \in C_{comp}.NT \Leftrightarrow c_v.outp_q \in c_v.NT$ holds).

[14] *FMERGE* is a function merging two functions with mutually exclusive domains to one that is defined on the union of these domains and preserves both original mappings.

- If the edge modeled by both $outp_q$ and inp_r is only attached to the pin linking them and to no other one, $act_{q,r}$ will be internal and is consequently added to $C_{comp}.Act_{int}$. Otherwise, it is an external action and added to $C_{comp}.Act_{ext}$.
- If $outp_q$ is not attached to another link, it is triggered according to the production rules. Thus, $act_{q,r}$ is triggered as well if it is an internal action. So, the synchronous coupling follows the cTLA/c constraint that internal actions must have triggers.

Besides of the process actions modeling the coupling of activities by input and output pins, the local activities A_0 to A_n may have also internal local actions and A_0 may have actions describing its links to the pins of the call behavior action in which it is defined. For each of these actions, a system action is defined in C guaranteeing that they are also executed in C. Moreover, these actions are added to $C_{comp}.Act_{int}$ or $C_{comp}.Act_{ext}$.

6.2 Asynchronous Coupling

The synchronous coupling of collaborations is possible whenever the actions that should be coupled are bound to the same collaboration role in the enclosing collaboration. This means that, in a component-oriented specification produced by model transformation, they can be implemented within the same state machine and hence be executed within the same state machine transition. For implementation purposes, however, we may want that also partitions bound to the same collaboration role may be realized by different state machines of the same component. A reason for that might be, for example, to let different parts of a collaboration be executed on different operating system threads to prevent long-running operations from blocking other behaviors. As passing events between different state machines is always buffered in our approach (see [20]), this implies an asynchronous coupling between the processes.

In this case, we can add a stereotype to call behavior actions that should be coupled asynchronously. For the cTLA/c model we assume that activities A_0 and A_i to be linked via buffers are not coupled directly but via a collaboration B modeling the buffering of tokens. B is specified by a cTLA process c_B similar to *Transfer* from Sect. 5.1. In contrast

to *Transfer*, however, both actions *send* and *receive* are associated to the same partition P in which the pin is located. Assuming k different output actions $outp_1, \ldots, outp_k$ and l input actions $inp_1, \ldots, inp_l$, the buffered coupling from activity v to activity w is specified by the $k+l$-many system actions assigned to C specified in the following (with $q \in \{1..k\}, r \in \{1..l\}$):

$$\left.\begin{array}{l} C.send_q(\ldots, ot : [ET \to TT], \ldots) \triangleq \\ \qquad c_v.outp_q(\ldots, ot, \ldots) \wedge\ c_B.send(ot(\text{"}e_o\text{"})) \\ C.receive_r(it : TT, \ldots) \triangleq \\ \qquad c_B.receive(it) \wedge c_w.inp_r(it, \ldots) \end{array}\right\} \begin{array}{l} v, w \in \{0, i\}, \\ v \neq w \end{array}$$

The l actions $C.receive_r$ have a trigger. In contrast, the actions $C.send_q$ are only trigger actions if the bound action $C_v.outp_q$ is also triggered (i.e., $C.send_q \in C_{comp}.NT \Leftrightarrow c_v.outp_q \in C_{el_v}.NT$ holds). The other settings of C_{send_q} and $C_{receive_r}$ in the cTLA/c tuple C_{comp} are similar to those of the synchronous case.

6.3 Asynchronous Multi-Session Coupling

To make our approach SPACE versatile for the development of real services, we must be able to deal with a number of different components providing identical functionality. For instance, the *Mobile Home Control* specification depicted in Fig. 7 is only useful if it specifies an arbitrary number of zone managers and telephones. To model several entities of a particular type, in the UML 2.0 collaborations the components may contain multiplicities (e.g, arbitrary many entities of the phone p and heater h as well as at least one entity of the zone manager z may occur). To achieve this multiplicity for the behaviors, cTLA/c may contain not only simple collaborations but also collaboration arrays each defining a whole number of identical simple collaborations. In cTLA/c, an collaboration array with multiplicity m corresponds to m-many simple collaboration instances each providing the same behavior. We express multiple occurrences of a collaboration by cTLA array processes as shown below for the zone session:

```
PROCESS ZoneSessionMult (m: INTEGER; ...)
ARRAY
  id: 1..m OF z: ZoneSession(...);
END
```

In general, this cTLA array operator (see also [8]) defines for each variable v of type $VType$ an array variable zXv of type $[1..m \to vType]$ keeping m-many values of v. Likewise, every action of z gets a new parameter $id : 1..m$ describing which occurrence of zXv is accessed. In this way, we can specify several concurrent sessions of a certain collaboration by one cTLA array process.

To determine the number of instances of a collaboration (i.e., the parameter m in the array process), we have to analyze the multiplicities of the participants bound to it. A meaningful solution is to provide a collaboration instance for each combination of participant instances. In the example, this is done for the activity *Zone Session*. If N_p is the number of phones modeled and N_z the number of zone managers, we create, as the other participating collaboration roles are not defined multiple, $N_p \cdot N_z$-many instances of the zone session collaboration[15]. For the activity *Temperature Update*, however, a similar determination of the number of instances would not be useful since one heater is only connected to one zone manager. Therefore it is sufficient to provide only on activity instance for each heater in the system.

On the other side, it could be useful to create more than one instance for every combination of collaboration roles. This is not possible to express in standard collaborations, as UML does not foresee multiplicities on collaboration uses. In these cases we therefore add a stereotype as part of our profile [15] to the collaboration use marking that it can be execute several times among the same participants. The multiplicity of the collaboration use is then multiplied with that of its participants, and the ID for the session takes the sequence number as an additional field.

Every activity referred by an call behavior action that represents a multi-session collaboration is formally modeled by one cTLA array process. For the activity describing the surrounding part of the call behavior actions, we face the problem that its partitions may have a different multiplicity. For instance in Fig. 3, we have one location server, N_z zone managers and N_h heaters. Thus, we cannot describe all partitions with a single activity process as in the singular case from Sect. 6.1. Instead,

[15] The formal existence of a session instance does not necessarily imply an ongoing behavior or demand real system resources, as an execution platform may instantiate needed state machines only when the behavior actually starts.

we define a separate activity process for every partition expressed by an cTLA/c tuple. Formally, the partitions representing a collaboration role with multiplicity larger than "1" are also specified by cTLA array processes.

The multiplicity of the collaborations has consequences for the coupling. In some cases we may still apply the couplings from the previous sections even if the originating collaboration role is multiple. This is the case if from a partition P only one activity instance A_i can be reached (e.g., it holds for the heaters, as each of them is only connected to one zone manager activity). In consequence, the decision how to traverse between P and the call behavior action hosting A_i is unambiguous and we can use the synchronous or asynchronous couplings discussed in the preceding subsections. In the activities, this situation is made visible by the lack of the shadow-like border around the call behavior action.

In contrast to that, shadow-like borders of call behavior actions highlight crossovers in which selections are necessary. This is generally only the case for flows into a call behavior action, as every outgoing flow is univocal due to the fact that an activity instance is non-ambiguously attached to an instance of all partition instances to which it is attached. For input flows, we use the special statement **select** [16] already mentioned in Sect. 2.4. This statement is attached to each flow entering a call behavior action when there are multiple sessions to choose from. The statement simply describes a list of filters that can be applied successively to find those collaboration instances that should be notified. Filters can access data within the token or within the variables of the current partition and may also depend on the data within the individual sessions that it chooses among, which is possible as they are implemented within the same component and only requires read-access.

In cTLA/c, we model this coupling by linking the two collaborations via a special collaboration C_s implemented in cTLA as follows:

```
PROCESS SelectActivity(m_c, m_p: INTEGER;
                       select: [AVT × TT → SUBSET {1..m_c}])
CONSTANTS PartId ≜ [{1..m_c} → {1..m_p}];
VARIABLES
  q : [1..m_c → QUEUE(TT)];
INIT ≜ ∀ i ∈ {1..m_c}: q[i] = EMPTY;
ACTIONS
  send(id: {1..m_p}; iav: AVT; it: TT) ≜
    q′ = [c ∈ {1..m_c} ↦
          IF (PartId[c] = id ∧ c ∈ select[iav,it])
          THEN APPEND(q[c],it) ELSE q[c]] ;
  receive(id: {1..m_c}; ot: TT) ≜
       q[id] ≠ EMPTY ∧ ot = FIRST(q[id])
    ∧ q′ = [q EXCEPT! id ↦ TAIL(q[id])];
END
```

The parameters of the process are m_c describing the number of instances of the activity A_i bound in the call behavior action, m_p as the number of instances for the partition P, and *select* as a function describing the select statement assigned to the input pin. In particular, *select* maps settings of the auxiliary variables in P and the signature of the token traversing through the pin to the set of the instance identifiers which should get a copy of the token. *PartId* is a function mapping the identifier of A_i's instance to those of P. The process has a queue for every instance of A_i as specified by the array variable q. The action *send* models the appending of tokens to the buffers according to the select statement. Its parameter *id* refers to the identifier of the partition instance. Of course, a token may only be send to instances of the activity in the call behavior action attached to P as expressed in the condition $PartId[c] = id$. Moreover, it has to follow the select statement as specified by $c : in : select[iav, it]$. The action receive describes the consumption of an element from the queue by an instance of A_i with the identifier *id*.

In consequence, *send* is coupled with each of the k actions $outp_1, \ldots, outp_k$ of the cTLA array process c_0m generated from c_0 modeling the flow e_o towards the input buffer. Here, the parameter *id* of the action created by the cTLA array operator is mapped to *id* in *send*. In the same way, the action *receive* is joined with the l-many actions $inp_1, \ldots, inp_l$ of the process c_im specifying the downstream flow e_i of the pin (with $q \in \{1..k\}, r \in \{1..l\}$):

$$C.send_q(id : 1..m_p; \ldots; ot : [ET \to TT]; \ldots; ov : [ET \to VT]; \ldots) \triangleq$$
$$\quad c_om.outp_q(id, \ldots, ot, \ldots, ov, \ldots) \wedge c_s.send(id, ot[\text{"}e_o\text{"}], ov[\text{"}e_o\text{"}])$$
$$C.receive_r(id : 1..m_c; it : TT, \ldots) \triangleq$$
$$\quad c_s.receive_r(id, it) \wedge c_im.inp(id, it, \ldots)$$

For flows through an output pin, we have, as mentioned above, no freedom to select an activity. Thus, this flow is specified by a simple buffer as introduced in Sect. 6.1.

6.4 Final System Model

After composing all cTLA/c representations of the UML activities with each other, we achieve a preliminary cTLA/c system description C_{pSys}. This model consists only of internal actions act_{pSys} each having a dedicated trigger. This reflects that all call behavior actions are properly bound and each pin has exactly one upstream and downstream link. In the activities, we model the interaction of a service with its environment (e.g., the service user functionality) by means of special signals expressed by send and receive nodes [17]. Formally, these signals are described by the action parameters *is* and *os* which model the sets of signals coming from resp. heading towards the environment.

To achieve the final cTLA/c system model C_{Sys}, we still have to specify the handling of the local auxiliary variables defined in the activities. This is done that lately in order to enable the access of all auxiliary variables defined for a component participant P from all activity partitions, the collaboration roles of which are assigned to P. To model the auxiliary variables, we use the process *AuxVar*:

```
PROCESS AuxVar (initavt: VT;
                navt: [ SUBSET ET × [ET → VT] → VT])
VARIABLES
  store: VT;
INIT ≜ store = initavt; ACTIONS
  access (current: VT; finedge: SUBSET ET; new: [ET → VT]) ≜
     current = store ∧ store′ = navt[finedge,new];
END
```

Here, the auxiliary variables are stored in a variable *store* which initially carries the values expressed by the process parameter *initavt*. The access to the store is modeled by the action *access* which in a single step accesses the current value of the auxiliary variables (with action parameter *current*) and stores the new one reflecting the atomicity of a cTLA action. The problem of handling auxiliary variables is that flows modeled by an action may be forked and the resulting downstream flows may pass different call operation actions which can create conflicting variable assignments. To solve this problem, we defined a process parameter *navt*. It is provided by the set of final edges in a flow and the corresponding variable settings from which a unique setting is computed. This function is applied in the action *access* to calculate the new value of the variable *store*.

For every single participant, we define an instance of *AuxVar* and for every multiple partition one of the corresponding array process modeling m_p instances of the partition variables. Every action act_{pSys_k} of C_{pSys} assigned to the partition P is linked with the action $P.access$ storing the auxiliary variables for the instance id of partition P to which act_{pSys_k} is assigned:

$$
\begin{array}{l}
act_{Sys_k}(id : 1..m_p; is, os : SUBSET\ TT) \triangleq \\
\exists : it_s, ot_s, iv_s, ov_s, last_s : \\
\quad act_{pSys_k}(id, it_s, ot_s, iv_s, ov_s, is, os, last_s) \wedge \\
\quad P.access(id, iv_s, last_s, ov_s)
\end{array}
$$

Besides the identifier id, the resulting system action act_{Sys_k} uses only *is* and *os* as parameters modeling that the external signals are the means to interact with the environment. The overall system specification C_{Sys} defines now the formal semantics of the full service model described by UML 2.0 collaborations and activities following the SPACE approach.

7 Concluding Remarks

We presented cTLA/c, a style of the compositional Temporal Logic of Actions that captures the behavior of collaborative system specifications. We think of cTLA/c foremost as a background technique to understand the formalism of collaborative specifications expressed in other languages, such as UML. For our approach SPACE, we use UML 2.0 collaborations in combination with activities, and have therefore presented how they can be transformed to cTLA/c specifications and in this way provide them with a non-ambiguous formal semantics. The provision of a formal semantics does not end in itself but, in our opinion, is a central ingredient for the automated development of high-quality software. It is the basis for meaningful semantic checks as, for instance, to be done with the model checking approach introduced in [29]. With such kind of methods, one can analyze collaborative service specifications thoroughly and ensure, for instance, that different views using distinct diagrams describe one consistent execution model. Another application for the formal semantics based on cTLA/c is model transformation. It provides us with the means to verify formally that transformation tools generate target models that fulfill the behavioral constraints of the source

models. As presented in [19], we checked that the transformation from UML activities to state machines is correctness-preserving.

In the moment, the service specifications in form of collaborations and activities are the most abstract ones that are used in our approach. Nevertheless, there may be further layers of abstraction in specifications. These specifications could consider collaborations on higher abstraction levels, which may be useful in early specification attempts when the complete system behavior or aspects of distribution are not yet known and must successively be elaborated. For that, notations like goal sequences [6, 27] or DisCo [22] may be useful. As cTLA resp. cTLA/c can also be used to specify such abstract models, we can carry out formal logic proofs to guarantee the correctness of the manual or automated refinement steps from these very abstract specifications to those used in SPACE. Thus, a complete formal and highly automated development of distributed services all the way from very abstract scenario-based descriptions to executable code will be feasible.

References

[1] Martín Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[2] R. J. R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. *Distributed Computing*, 3:73–87, 1989.

[3] Rolv Bræk. Unified System Modelling and Implementation. In *International Switching Symposium*, pages 1180–1187, Paris, France, May 1979.

[4] R. J. A. Buhr and R. S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, Inc., 1996.

[5] Humberto Nicolás Castejón and Rolv Bræk. A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. *ICSE's 5th Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06)*, 2006.

[6] Humberto Nicolás Castejón and Rolv Bræk. Formalizing Collaboration Goal Sequences for Service Choreography. In Elie Najm and Jean-François Pradat-Peyre, editors, *26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *Lecture Notes in Computer Science*. Springer, September 2006.

[7] Jacqueline Floch. Supporting Evolution and Maintenance by Using a Flexible Automatic Code Generator. In *Proceedings of ICSE-17 – 17th International Conference on Software Engineering*, Seattle, April 1995.

[8] Peter Herrmann. *Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen*. PhD thesis, Universität Dortmund, 1997. In German.

[9] Peter Herrmann and Frank Alexander Kraemer. Design of Trusted Systems with Reusable Collaboration Models. In Sandro Etalle and Stephen Marsh, editors, *IFIP International Federation for Information Processing*, volume 238, pages 317–332. IFIP, Springer, 2007.

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

[11] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, 2002.

[12] ITU-T. *Recommendation Z.120: Message Sequence Charts (MSC)*, 2004.

[13] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 342–416, London, UK, 1991. Springer-Verlag.

[14] Frank Alexander Kraemer. The Ramses and Arctis Tools. http://www.item.ntnu.no/∼kraemer/tools.

[15] Frank Alexander Kraemer. UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097,

Department of Telematics, NTNU, Trondheim, Norway, March 2007.

[16] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In E. Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, 2007.

[17] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[18] Frank Alexander Kraemer and Peter Herrmann. Semantics of UML 2.0 Activities and Collaborations in cTLA. Avantel Technical Report 3/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, September 2007.

[19] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7, 2007.

[20] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[21] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems*. Springer, 2005.

[22] Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of Distributed Cooperation, their Refinement and Implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, April 1998.

[23] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[24] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.1, February 2007. formal/2007-02-03.

[25] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *Current Trends in Concurrency. Overviews and Tutorials*, pages 510–584, 1986.

[26] Judith E. Y. Rossebø and Rolv Bræk. Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition. In *Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06)*, pages 206–215. IEEE Computer Society Press, 2006.

[27] Richard Torbjørn Sanders. *Collaborations, Semantic Interfaces and Service Goals: a way forward for Service Engineering*. PhD thesis, Norwegian University of Science and Technology, 2007.

[28] Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[29] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[30] Eirik A. M. Vefsnmo. DASOM — A Software Engineering Tool for Communication Applications Increasing Productivity and Software Quality. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 26–33, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[31] Chris A. Vissers, Guiseppe Scollo, Marten van Sinderen, and Hendrik Brinksma. Specification Styles in Distributed System Design and Verification. *Theoretical Computer Science*, 89:179–206, 1991.

[32] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

About the Authors

Frank Alexander Kraemer has a master's degree (M.Sc.) in Information Technology and a diploma (Dipl.-Ing.) in Electrical Engineering from the University of Stuttgart, Germany. After finishing his master's thesis at the Department of Telematics at the Norwegian University of Science and Technology (NTNU) in 2003, he joined the networked systems research group. Since then, Frank is working on his doctoral thesis with Professor Rolv Bræk and Professor Peter Herrmann as supervisors.

Peter Herrmann studied Computer Science at the University of Karlsruhe, Germany, and achieved his diploma in 1990. From 1990 to 1999 and from 2001 to 2005 he worked as a researcher at the University of Dortmund, Germany, and did his doctorate in 1997 on problem-oriented correctness-guaranteeing design of high-speed communication protocols. Since 2005, he is professor on Formal Methods at the Department of Telematics (ITEM) of the Norwegian University of Science and Technology (NTNU) in Trondheim. Peter works in the areas of formal specification, design, implementation and verification of distributed systems, networked services and continuous-discrete technical systems, functional and security aspects of distributed component-structured software, and trust management. He is author or co-author of more than 50 journal and conference papers and editor of a special journal issue on security and trust in electronic commerce as well as of the proceedings of the 3^{rd} International Trust Management Conference. In 1999, he was awarded a stipendium for two years as a postdoctoral researcher in the postgraduate research program "Modelling and Model-Based Design of Complex Technological Systems" of the University of Dortmund. He is a member of the IFIP Working Group 11.11 on Trust Management.