Article:

Cavalcanti, Ana Lucia Caneca orcid.org/0000-0002-0831-1976, Filho, Madiel Conserva, De Oliveira Salazar Ribeiro, Pedro Fernando orcid.org/0000-0003-4319-4872 et al. (1 more author) (2023) Laws of Timed State Machines. The Computer Journal. bxad124. ISSN 1460-2067

https://doi.org/10.1093/comjnl/bxad124

The Computer Journal, 2023, 1–42
https://doi.org/10.1093/comjnl/bxad124
Original Article

Laws of Timed State Machines

Ana Cavalcanti[1,*], Madiel Conserva Filho[2], Pedro Ribeiro[1] and Augusto Sampaio[2]

[1]University of York, UK
[2]Centro de Informática, Universidade Federal de Pernambuco, Brazil
*Corresponding author: Ana.Cavalcanti@york.ac.uk

State machines are widely used in industry and academia to capture behavioural models of control. They are included in popular notations, such as UML and its variants, and used (sometimes informally) to describe computational artefacts. In this paper, we present laws for state machines that we prove sound with respect to a process algebraic semantics for refinement, and complete, in that they are sufficient to reduce an arbitrary model to a normal form that isolates basic (action and control) elements. We consider two variants of UML-like state machines, both enriched with facilities to deal with time budgets, timeouts and deadlines over triggers and actions. In the first variant, machines are self-contained components, declaring all the variables, events and operations that they require or define. In contrast, in the second variant, machines are open, like in UML for instance. Laws for open state machines do not depend on a specific context of variables, events and operations, and normalization uses a novel operator for open-machine (de)composition. Our laws can be used in behaviour-preservation transformation techniques. Their applications are automated by a model-transformation engine.

Keywords: UML; robotics; verification; normalization; CSP; refinement

1. INTRODUCTION

In both the industrial and research communities, state machines are widely used [1–3] to record and convey designs and simulations of (embedded) control software [4]. For long now, state machines have also been incorporated in more general modelling notations, notably, the very popular UML [5] and SysML [6].

In this paper, we present laws of state machines. We consider two different notations: one for self-contained components that define behavioural models in the context of identified variables, events and operations, and one for open machines, whose context is defined in other components of the models, such as (active) classes. Self-contained components can be analysed in isolation, and are convenient for compositional verification. Open machines are akin to those adopted in UML, and support a flexible approach to modelling. Verification of an open machine typically needs to be in context, rather than compositional.

For decades, laws [7] have been recognized as useful results to support reasoning about programs [8], designing correct compilers [9–11] and, when interpreted as program transformations, supporting informal programming practices such as refactoring [12–14]. Laws of Occam [15] capture useful properties of concurrency and communication. Laws of functional programming are elegantly addressed in [16]. Laws of logic programming are presented in [17]. Laws of object orientation can be found in [18], and for a variant of Java for safety-critical systems in [19]. Laws of hardware synthesis are the subject of [20].

In all these works, a normal form is used to attest the expressiveness of a proposed set of laws by establishing a relative notion of completeness via an associated strategy for normalization. For example, a (relative) notion of completeness for laws of concurrent operators can be established by showing that the laws are powerful enough to reduce arbitrary concurrent programs to a sequential program that uses a restricted subset of the language constructs, that is, a normal form. This is done, for instance, for Occam [15]. Similarly, the purpose of the laws in [18] is to capture algebraic properties of object-oriented constructs. By showing that an arbitrary program in an object-oriented language like Java can be reduced, using the laws, to an imperative subset, that is, to a normal form, a measure of the comprehensiveness of the laws is provided.

For state machines, there are numerous formal semantics. We can find formalizations using tailored semantic domains [21–23], graph transformations [24], programs [25] and process algebra [26, 27]. Besides UML machines, there are semantics for SysML [28] and Stateflow [29]. For a very specific notion of state machine, used to represent data structures and types in a program context, there is a seminal calculus [30]. We are, however, not aware of a set of algebraic laws for a state-machine notation that includes constructs to define time properties, and that have been proved sound using an independent denotational model.

Both our self-contained and open machines define behaviour in terms of variables, events and operations. Events represent interactions (via sensors and actuators, for example). Operations represent computational mechanisms potentially defined by further machines. In both notations, state machines can be hierarchical, and the action language is well-defined, including extra time constructs, not available in UML, for modelling of temporal properties: budgets, timeouts and deadlines. Well-formedness conditions rule out inter-level transitions following accepted good practice [31].

A self-contained state machine encapsulates a declaration of a context of variables (local and required), events and required

Received: September 30, 2023. **Revised:** August 15, 2023. **Accepted:** November 27, 2023

operations that can be used in its definition. In concrete terms, we consider the state-machine notation adopted in a domain-specific language for robotics called RoboChart [32]. In particular, we consider the RoboChart constructs to specify time budgets, timeouts and deadlines, over triggers and actions, and also use of clocks, to capture time properties of control designs.

A number of domain-specific languages have been proposed [33]. RoboChart is distinctive in its semantics, and support for (automatic) verification of design properties [34]. The semantics is defined in a timed variant *tock*-CSP of the process algebra for refinement CSP [35]. We use this semantics in the work presented here to prove soundness of our laws.

An open state machine can be used in any context where their elements are, or can be regarded to be, in scope. In this respect, they are similar to UML state machines, but we retain a well-defined action language, timed constructs, and rule out inter-level transitions. For open machines, we define in this paper a *tock*-CSP semantics to support the proof of soundness of the laws.

Our primary contribution here is two sets of laws, for RoboChart and for open machines. The notion of equality is that embedded in the notion of refinement in *tock*-CSP. Precisely, equality indicates that the processes for the equated terms refine each other, so that the diagrams define the same behaviour in terms of possible timed interactions and deadlocks. Here, an interaction corresponds to a required variable access, event occurrence or operation call. For machines that define operations, equality considers all contexts in which the operation can be called.

Additional contributions are a compositional account of RoboChart semantics, a novel semantics for open state machines, proof of soundness of the laws and notions of completeness for the two sets of laws. In each case, we define a normal form that characterizes a machine whose control structure embedded in (hierarchical) states, and actions, including those involving timed statements, is revealed. For that, we use operation calls to replace actions, and, in the case of open state machines, a novel machine combinator.

These normal forms do not flatten the structure of a model (as usual in works on algebraic semantics). Instead, they isolate the action and control flow constructs, so that a machine is expressed using a small number of primitive patterns. This means that model transformation techniques (for refactoring or translation to other notations, for instance) can be significantly simplified to consider just these patterns. A normalization strategy establishes that our laws are enough to normalize any RoboChart or open machine.

Next, we give an overview of our notations for RoboChart and open state machines. Our normal forms are defined in Section 3. The laws are presented in Section 4 as part of the description of normalization procedures. Evaluation of our work comes in three forms: in Section 5 we present examples and a tool that mechanizes our laws and normalization strategies, and in Section 6 we describe our proofs of the soundness of the laws. We conclude in Section 7.

2. OUR MACHINE NOTATIONS

The core notation for the state machines considered here is, by far and large, standard. To illustrate the constructs, we present a model for the system in [36]: an efficient robot to harvest apples in an orchard in which the tree branches are trained along a trellis. We capture the algorithm in [36]. It seems to have some limitations, but given the use of an informal notation for its description, there may be ambiguities in the description rather than in the actual implemented algorithm. It is not our objective, however, to redesign the algorithm; to illustrate the use of our notations and laws, a faithful account of the work in [36] is more interesting. (We leave it as future work to analyse the application and possibly produce a modified design.)

The robotic platform for the harvester is an arm, with a custom manipulator and end-effector with six degrees of freedom. To define a self-contained component that models its control software, we define in Fig. 1 some interfaces. First, the interface ArmOperations declares operations that represent facilities of the platform to move the arm to various positions and to manipulate the apples. The system also includes a camera; it is represented by an event takePic in the interface Camera. This (input) event communicates an image, represented here as an element of a type Image, whose definition (omitted in Fig. 1) simply gives its name. The complete RoboChart model is available[1] .

Additional interfaces in Fig. 1 define variables, events and operations that are not provided by the platform. These extra variables are either local to machines or shared among machines. The extra events are used for communication between machines, rather than with the platform. Finally, the extra operations are implemented for the application, rather than embedded in the platform. In the example, the extra interface GlobalVariables in Fig. 1 declares two variables. We have a record of the set of apples found in the current image via their coordinates in 3D. For each of them, we also record a tuple with three positions: these are the joint positions (of type JointPos) for the arm to approach, pick and store the apple. The values of these variables are defined with the help of an operation CHTBA(), which models a Vision algorithm that uses Circular Hough Transformation (CHT) and blob analysis to identify the apples. Using positions, a TravellingSalesman algorithm defines the NearestNeighbour() apple.

The interfaces SolverControl and GoHomeControl declare events used to control the flow of execution. Populating positions using an inverse kinematic solver, and manoeuvring the arm to a home position occur in parallel. The events control the forking.

Finally, the interfaces TimeConstants and Locations declare constants used to specify time properties of the control software, and the home and store positions of the arm (homePos and storagePos).

Figure 2 shows a RoboChart state machine AppleHarvestControl for the harvester control software. This component uses the interfaces in Fig. 1 to declare its required (®) variables and operations, and the events and local constants that it defines (ⓘ). In addition, AppleHarvestControl declares three local variables img, localized and nextApple. In what follows, we describe how the machine uses all the elements in this context.

A state machine has a unique initial junction (represented by a black circle with an i). In AppleHarvestControl, it has a single transition out of it into the state Prepare. States have entry, during and exit actions, executed when the state is entered, while it is active, and when it is exited. In Prepare, the entry action first calls the platform operation hideArm() to take the arm out of the way of the camera.

In sequence (;), we have a wait statement, which can be used to define a time budget: an amount of time to wait, that is, pause, before proceeding. In this case, this is an amount of time between 0 and hideTime time units, which, in an implementation, is used to allow the effect of the call hideArm() to take place, and the arm to position itself out of the way. The nondeterminism in the time budget indicates that an implementation may allow for almost no

[1] robostar.cs.york.ac.uk/case_studies/

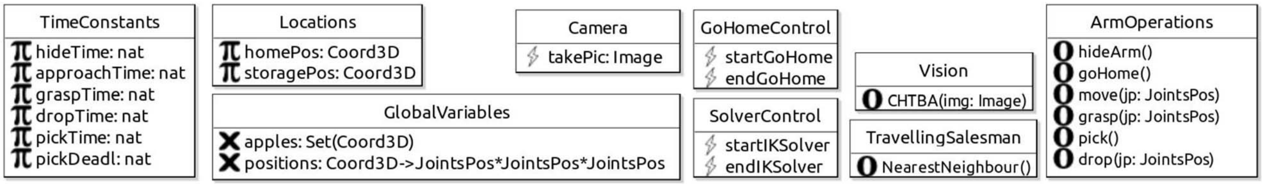


Figure 1. Data model for the apple harvester.

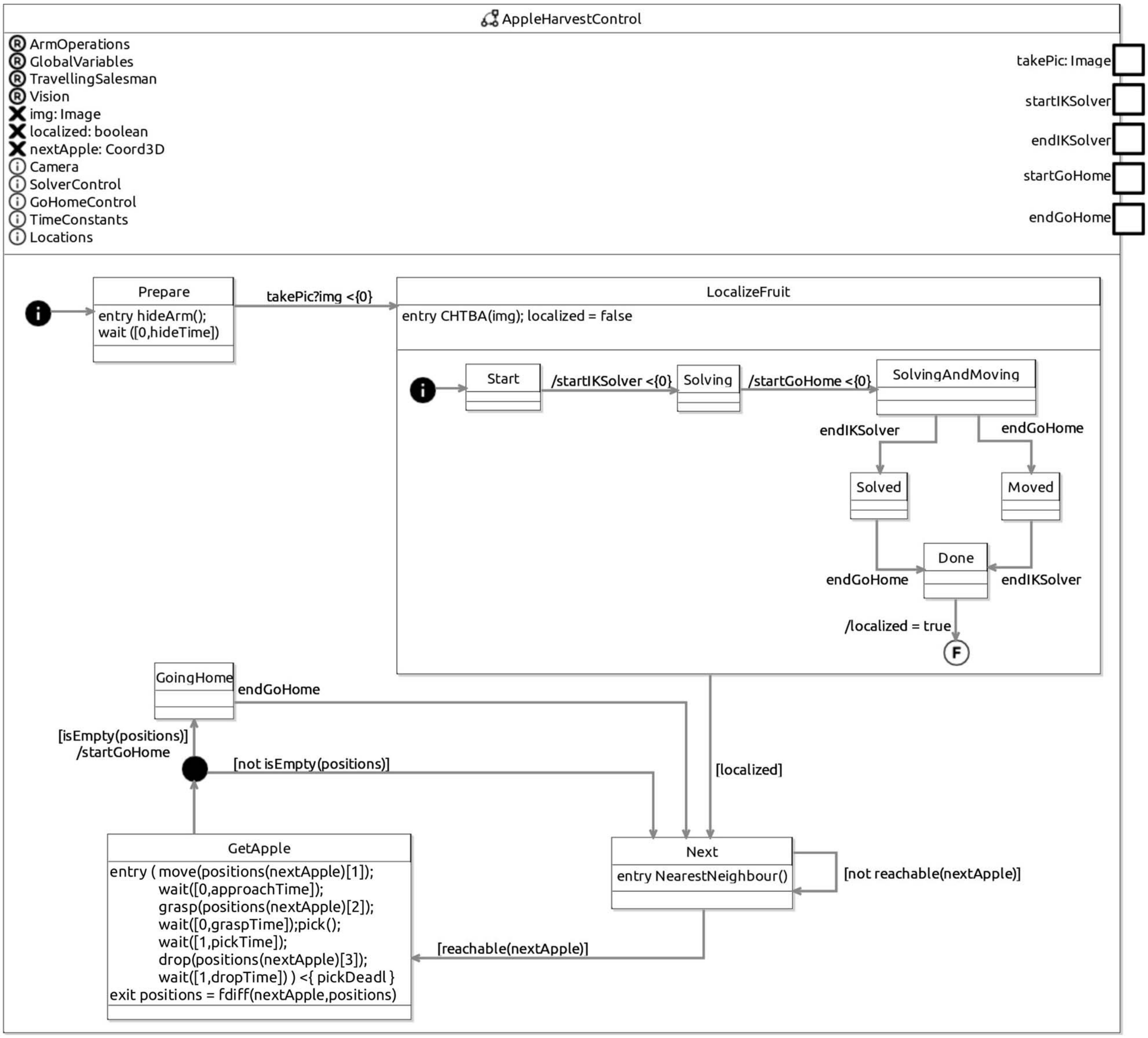


Figure 2. Main state machine for the apple harvester.

time, if the arm is very fast, or is already hidden, for example, or take up to hideTime time units. Here, hideTime is a constant whose value is not defined in the model, and depends on the specific arm and environment design.

Once a state is entered, the transitions out of it become available if their guards, if any, hold and their trigger events, if any, can occur. For Prepare, its single transition has a trigger: takePic. Events take place when a communication via the connection with this event is available. For takePic, the connection is ultimately with the platform to receive an image from the camera. Since connections with a platform are normally asynchronous, it is expected that an image is immediately available. This is recorded by a deadline 0 (<{0}) on the trigger. The input image is recorded in the local variable img.

The transition from Prepare leads to a composite state LocalizeFruit. It has an entry action that calls the software operation CHTBA(), which updates the global variable apples. The entry action also assigns false to a local variable localized that records whether the localization effort is concluded.

A composite state has itself a state machine that defines behaviour that takes place while the composite state is active. If the composite state has a during action, it takes place in parallel with the behaviour of the machine. In our example, localization involves defining the positions of the joints to deal with the

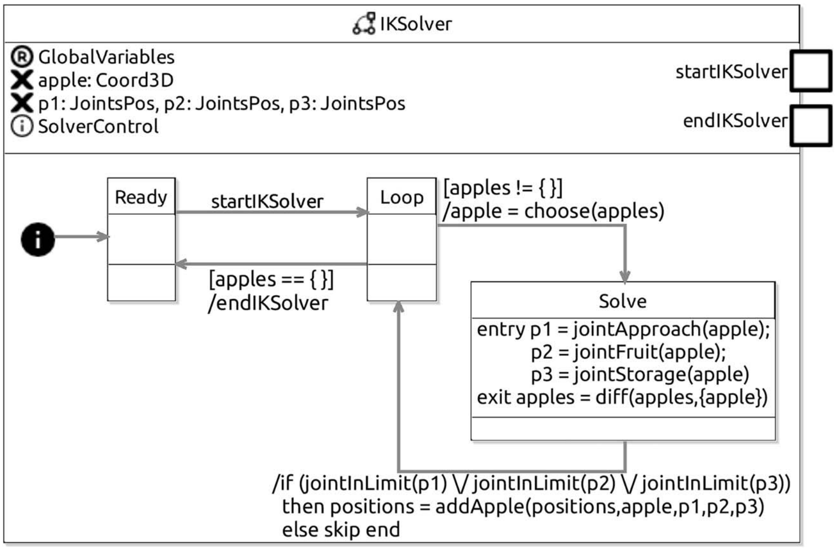


Figure 3. IK solver of the apple harvester.

apples found, and moving the arm to the home position. This is achieved by the machine in LocalizeFruit. Its initial junction leads to a state Start, with a transition that has no guard or trigger, and so is immediately taken. This transition has an event action, startIKSolver, which takes place immediately. This event (declared in the interface SolverControl) is used to communicate with the machine IKSolver in Fig. 3.

The behaviour defined by IKSolver is initiated upon occurrence of the event startIKSolver. This machine provides an inverse kinematics solver to obtain joint solutions for three positions: the approach, the fruit and the storage positions, which are recorded in the variable positions (declared in the interface GlobalVariables). Once startIKSolver happens, IKSolver goes to a state Loop. If the set of apples is not empty (apples != {}), a transition to the state Solve takes place. The action of this transition assigns to a local variable apple a value chosen from apples, defined by a function chosen (whose definition is omitted here).

In Solve, three functions, jointApproach, jointFruit and jointStorage, are used to calculate joint positions and assign them to local variables p1, p2 and p3. The apple is removed from apples (using the set difference operator) and then control moves back to Loop. In the transition back, the action is a conditional. It checks if the joint positions are all feasible (in the limits of the joint abilities and of the workspace). If so, they are added to positions using a tailored (omitted) function addApple. Otherwise, nothing is done, as defined by the action skip. So, the positions are discarded.

When there are no more apples (apples == {}), control moves back from Loop to the state Ready, where the machine IKSolver waits for the next startIKSolver event. In moving to Ready, IKSolver communicates with AppleHarvestControl using the event endIKSolver, to indicate that it has finished its work.

In AppleHarvestControl, in the composite state LocalizeFruit, after the communication via startIKSolver, in the state Solving, another transition communicates with a machine GoHome via the event startGoHome. That machine, omitted here, uses the platform operation goHome to move the arm to the home position. Now in the state SolvingAndMoving, end signals endIKSolver and endGoHome from IKSolver and GoHome are accepted in either order. When both occur, localized is set to true. Now, the transition out of LocalizeFruit is enabled and the state Next is entered.

The behaviour of AppleHarvestControl after entering Next is defined using constructs already explained.

There are two forms of open machine. A basic open state machine can be just like a RoboChart machine, but it does not have declarations. For example, by removing the declarations of the RoboChart machines in Figs 2 and 3, we obtain basic open state machines. As already mentioned, that context is defined elsewhere in a complete model. Open machines, however, can also be defined by a combination of other open machines. In this case, the composed machines are fragments of a state machine that refer to other fragments, and composing all those fragments produces a basic open state machine. In this way, control flow, besides being embedded in actions and transitions, is also defined at the machine level. Such an operator facilitates transformation of machines, by allowing us to decompose a machine into smaller components that can be rearranged in a stepwise fashion.

We do not necessarily suggest that behavioural models are defined in this way. The combinator for open state machines, however, is useful in defining a normal form with a small number of machine patterns as described in Section 3. Equally, it can be useful to combine models for components of larger granularity.

Figure 4 presents an alternative model for the inverse kinematic solver that is defined by the combination of two open machines. The first, on the top, has a transition to a connecting node indicated by <Solve>. This is a reference to a node (state or junction) that, in a complete model, is defined in another machine. Connection nodes in open state machines resemble connection point references of UML, but connect different state machines.

In the second machine, the connecting state Solve is defined as an entry point, as indicated by the [⓪] inside its block. This machine also has a connecting node, <Loop>, that refers back to an entry point of the first machine. The machines are combined via the ⊙ operator, which matches the connecting nodes of one machine to the entry points of the other.

A connecting node cannot be the source of a transition, just the target. Junctions, however, can be an entry point for a connecting node, and, therefore, in open machines, they are named. In addition, a connecting node does not need to be defined as an entry point in the other machine. For example, the second machine in Fig. 4 did not need to have a definition for Solve. If this were

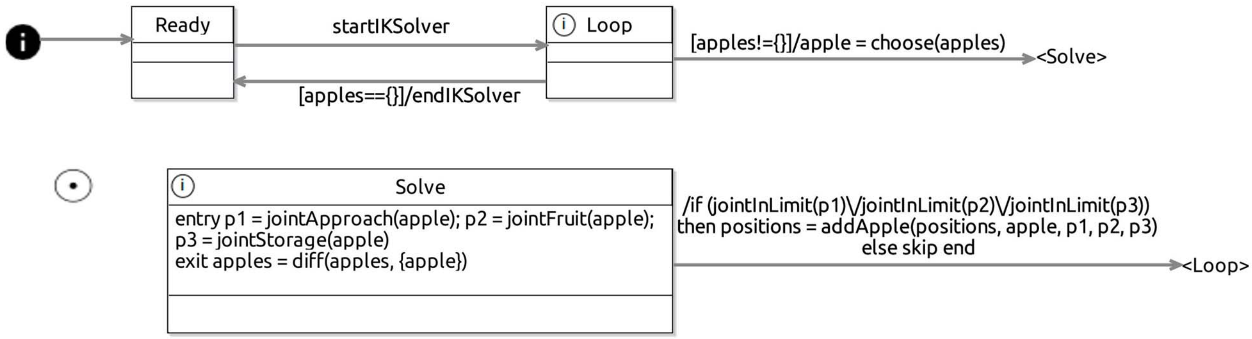


Figure 4. Open machine for the IK solver.

the case, however, combination with other machines would be required to define a complete model, where all connecting nodes have one definition.

We next describe our normal forms (and normalized versions of AppleHarvestControl and IKSolver).

3. NORMAL FORMS

To structure the presentation of our laws, and to equip them with a notion of completeness, we define in this section normal forms for RoboChart self-contained state machines (in Section 3.1) and open machines (in Section 3.2). In Section 4 we present our laws and their use in strategies to normalize models.

3.1. RoboChart normal form

A RoboChart model is normalized if every state machine in that model is in normal form, whether it is used to define behaviour for a controller, such as AppleHarvestControl, IKSolver and GoHome (omitted here) in our example, or to define operations, such as CHTBA and NearestNeighbour (omitted).

In Fig. 5, we provide the definition of the RoboChart metamodel for state machines. Classes depicted in grey are abstract; attributes whose names are written in italics are inherited; and those in bold face are compulsory. We are here concerned with a StateMachineBody, which can be a StateMachineDefinition or an OperationDefinition. AppleHarvestControl and IKSolver are examples of StateMachineDefinitions. An instance of OperationDef specifies an operation that may be called from a machine; that specification can itself be a state machine. As shown in Fig. 5, however, in contrast with a StateMachineDefinition, an OperationDefinition can have parameters. In our example, CHTBA, for instance, has a parameter of type Image.

A StateMachineBody is a NodeContainer with nodes, that is, States and Junctions, and transitions. A State can be Final, and a Junction can be Initial. Like in UML, a Junction is a decision point, where, in contrast with a State, the control flow does not pause. An Initial state is a Junction because, at the start, a transition from the Initial state to a proper State is immediately taken.

A StateMachineBody defines also a Context, which is a self-contained component. For that it records the variableList, events, and clocks local to the state machine, and its required operations. Variables, events and operations can also be declared via interfaces: with required variables and operations (rinterfaces) or defined (local) variables and events (interfaces).

States may have actions: EntryActions, DuringActions and ExitActions. A State is also a NodeContainer, since a composite state contains nodes and transitions of its own. Transitions connect two nodes: a source and a target. They may be triggered by a Communication, guarded by a condition, and contain an action that is executed when the transition is taken. We can also specify a deadline for a transition and reset a clock when the transition is taken. The clock is reset when the trigger occurs and the condition is true.

A ClockReset is a Statement. Every Action also has a statement. The metamodel for Statements is presented in Fig. 6. These include the usual Assignment, operation Call, sequence (SeqStatement) and conditional (IfStmt) statements. We also have Wait statements, and TimedStatements, which impose a deadline for the termination of a statement (stmt). They are both illustrated in Fig. 2. A communication statement (CommunicationStmt) identifies a communication via an event. Finally, a ParStmt is a parenthesized statement, needed to define scope for deadlines.

The Expression language is not surprising, but includes a construct sinceEntry(S) to denote the time since a state S has been entered. It avoids the need to declare and control a clock to account for that time, and is particularly useful when S is a composite state. In this case, entering S may involve an elaborate control flow that complicates the identification of the points in which the clock would need to be reset.

The normal forms for StateMachineDefinitions and OperationDefinitions defined below impose different restrictions on their StateMachineBody. Below, we consider first StateMachineDefinitions.

> **Definition 3.1.** (Normal form for StateMachineDef). A normalized StateMachineDefinition is specified by a StateMachineBody that is a NodeContainer that satisfies **both** conditions NCNF1 and NCNF2 in Fig. 7.
> NCNF1 and NCNF2 are defined in terms of the metamodel in Fig. 5. With those restrictions, we ensure that the data manipulations and the time control of the machine, normally defined in actions, are all encapsulated in operations, which are called in the actions and transitions of the StateMachineDefinition. NCNF1 and NCNF2 apply to all Statements and Transitions occurring anywhere in the StateMachineBody, including those in the nodes and transitions of the composite States.

In Fig. 8 we present a normalized version of the IKSolver state machine for the harvester example in Fig. 3. In this version, all actions are calls to operations, such as normal˙IKSolver˙t2˙op() and normal˙IKSolver˙entry˙op(), defined elsewhere and required by the normalized IKSolver via new interfaces, such as I˙normal˙IKSolver˙t2˙op() and I˙normal˙IKSolver˙entry˙op(), declaring operations.

The StateMachineBody of a normalized OperationDefinition can satisfy different restrictions.

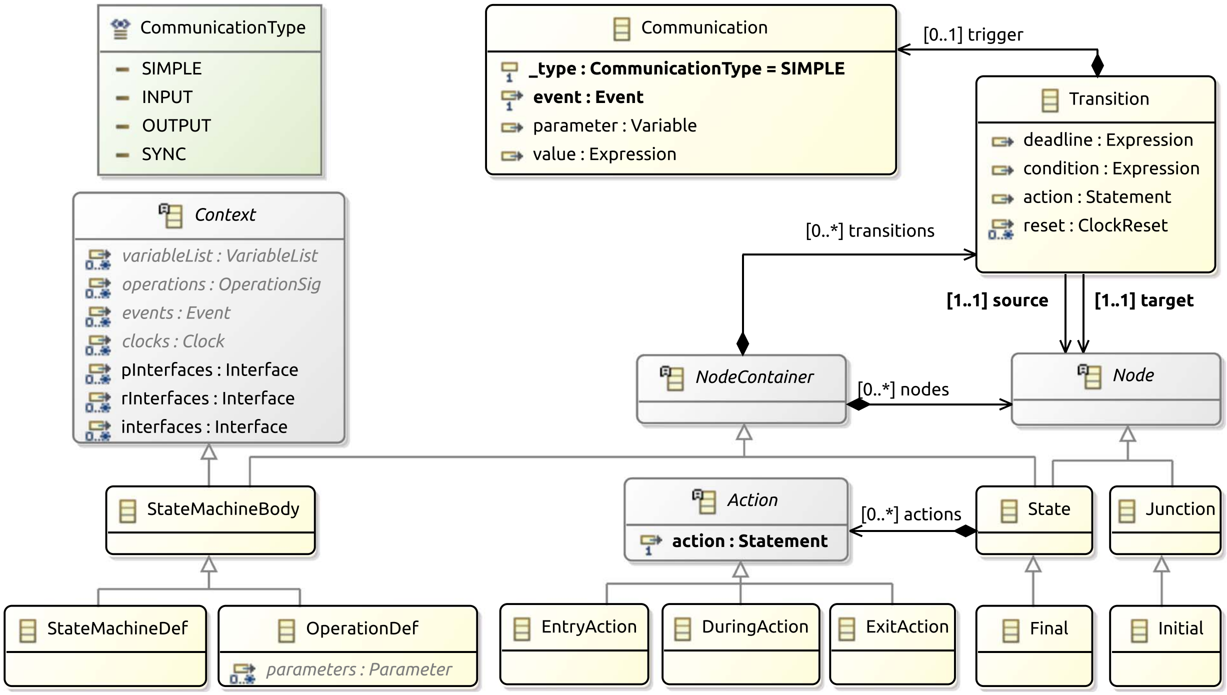


Figure 5. Metamodel for state-machine bodies, defining controller behaviour or operations.

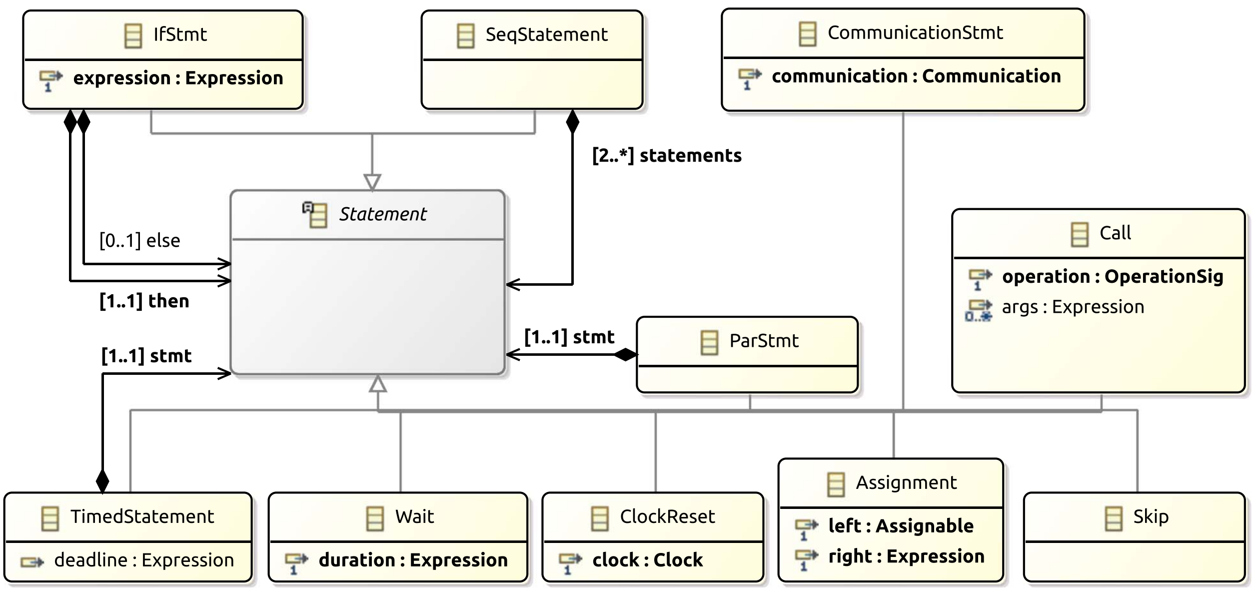


Figure 6. Metamodel for statements.

NCNF1 Every Statement in an Action or Transition is an operation Call.
NCNF2 In every Transition, the optional deadline is not present: it is null, and there is no use of a sinceEntry expression.

Figure 7. Normal form for a NodeContainer—a normalized NodeContainer satisfies **both** conditions.

Definition 3.2. (Normal form for OperationDef). A normalized OperationDefinition is specified by a StateMachineBody that is a NodeContainer that satisfies either OPDNF1 **or** OPDFN2 in Fig. 9.

With the normalization condition OPDNF2, we cater for OperationDefinitions whose bodies are normalized in the sense already specified for StateMachineDefinitions.

OperationDefinitions that satisfy OPDNF1 instead are called basic. To define the restrictions that are satisfied by basic operations, we use the notion of an action operation. This is an OperationDefinition whose set of nodes includes just an Initial junction and a Final state, and whose set of transitions includes just one Transition between them with an optional action (and no other element). OPDNF1 allows for an action operation whose Statement encapsulates one data or time statement, but no control flow: no conditionals or sequences, a limited form of deadline or a clock reset.

Figure 10 presents the definition of normal˙IKSolver˙t2˙op() used in Fig. 8. Its only action is an assignment, originally in the transition from Loop to Solver in the machine IKSolver in Fig. 3.

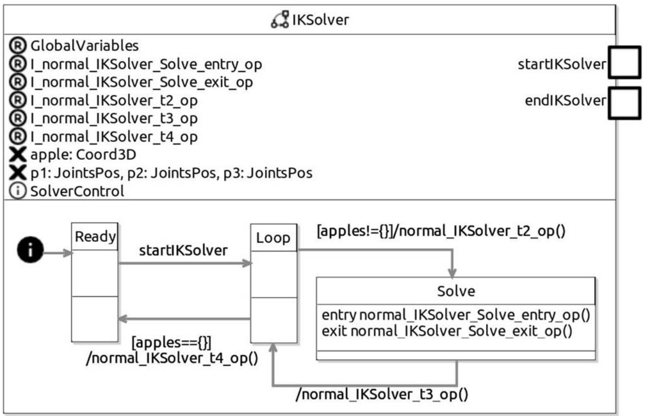


Figure 8. IK solver of the apple harvester—normalized.

OPDNF1 It is an action operation whose Statement is an Assignment, Skip, a CommunicationStmt, a TimedStatement of the form tStop() <{d}, for any integer expression d, or a ClockReset.
OPDNF2 The conditions NCNF1 and NCNF2 hold.

Figure 9. Normal form for an OperationDef—a normalized OperationDef satisfies **one** of these conditions.

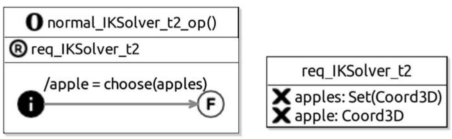


Figure 10. Basic operation in the normalized IK solver.

So, in summary, in a normalized RoboChart model, StateMachineDefinitions use operation calls for data and time services. And the OperationDefinitions themselves are either basic or also use calls to further operations that provide data and time services.

OPDNF1 also ensures that a deadline is only imposed on a call to the operation tStop() (see Fig. 11), which is itself normalized according to OPDNF2. This operation simply deadlocks and never terminates.

We can use this restricted form of deadline to express any deadline. First of all, we can indirectly impose a deadline on a trigger by effectively imposing it on an action, and making the trigger the only way to terminate that action within the deadline. For instance, in Fig. 2, we have a deadline 0 on the communication takePic?img used as a trigger on the transition from Prepare to LocalizeFruit. We can instead record that deadline using a during action tStop() <{0} in Prepare that imposes the deadline 0 on the termination of tStop(). Since this Statement does not terminate, to satisfy that deadline, we have to leave the state Prepare to interrupt its during action. So, the transition out of Prepare has to be enabled in time: its event trigger needs to occur with deadline 0, as required. The during action tStop() <{0} can be specified as a call to a (normalized) operation that satisfies OPDNF1. This approach works for all values of deadline, not only 0 as in the example.

Similarly, deadlines on an action can be imposed by having it in parallel with tStop()<{d}, and making termination of that action the only way to terminate tStop() within the deadline. For instance, in the machine in Fig. 2, we have a deadline 0 on the action startIKSolver in the transition from Start to Solving in LocalizeFruit. We can instead call the operation deadlineAction(0) whose definition is shown in Fig. 12.

The operation deadlineAction(d:nat) executes startIKSolver, which is the action of a transition that is reached as soon as deadlineAction(d:nat) is started, in parallel with tStop() <{d}, which is in a during action of a (composite) state S1, also reached as soon as the operation is started. Here, d is the parameter for deadlineAction(d:nat), defined as 0 in the call. Since tStop() cannot terminate and meet the deadline, the state S1 has to be exited for its during action to be interrupted and the deadline to be met. For that, the transition out of S1 has to be taken, and so the guard g of that transition has to hold. Since g is a local variable initialized to false, the transition to the final state of S1 has to take place, and the action startIKSolver terminate, so that g is assigned true and the transition is enabled. So, the deadline is (indirectly) imposed on startIKSolver. Since deadlineAction(d:int) uses tStop(), its definition requires the interface ItStop that declares this operation.

Normalization of deadlineAction(d:nat) requires just replacing its actions with calls to basic operations that perform them. In Section 4, we describe how any machine can be transformed (using our laws) to use just the restricted form of deadline tStop() <{d}.

A basic operation cannot include Wait statements. Instead, it can call either of the operations waitOp(i:nat) or waitInterval(m:nat,n:nat) in Fig. 11. They are both normalized according to OPDNF2. They both use a clock C, which is reset (#C) at the start, to encode a wait period. For that, waitOp(i: nat) has a state Waiting with a single transition to a final state. The guard on that transition uses a since(C) expression to require that it is taken only once the clock has recorded the passage of i time units. So, when in the state Waiting, the control flow pauses for i time units, since the only transition out of Waiting requires that i time units pass.

In the case of waitInterval(m:nat,n:nat), it uses a call waitOp(1) to pause one time unit. An interface IwaitOp declares waitOp(i:nat), and waitInterval(m:nat,n:nat) declares IwaitOp as a required interface. In the machine for waitInterval(m:nat,n:nat), two transitions out of a junction encode the nondeterminism of a wait([m,n]) statement. Once m time units have passed (that is, since(C) >= m), a transition may be taken to the final state, so that waitInterval(m:nat,n:nat) may terminate. While n time units are not over (since(C) < n), however, another self-transition is enabled that allows another time unit to pass: waitOp(1) instead of terminating.

Our normal form for open machines, defined next, enforces the above restrictions on node containers, but allows further structure in the construction of the body.

3.2. Open-machines normal form

The metamodel for open state machines is shown in Fig. 13. An OpenStateMachine can be basic (BasicOpenStateMachine) or composite (CompOpenStateMachine), that is, defined using ⊙. A well-formedness condition ensures that a BasicOpenStateMachine has at most one InitialState. Similarly, at most one of the left and right machines of a CompOpenStateMachine has an Initial state.

A BasicOpenStateMachine is similar to a StateMachineBody from RoboChart's metamodel, in that it is a NodeContainer (but not a Context). As such, it has nodes and transitions. The classes in Fig. 13 marked with an arrow on the top right-hand corner are those already presented in Fig. 5. We have, in this context, however, a new form of Node, namely, a NodeNameRef, whose attribute ref is the identifier of a node that is (expected to be) defined in another machine.

The states and junctions, including the initial junctions and final states, are similar to those of RoboChart, but have an extra boolean attribute, which indicates whether the Node is an entrypoint. We have classes EState, EFinal, EJunction and EInitial, which

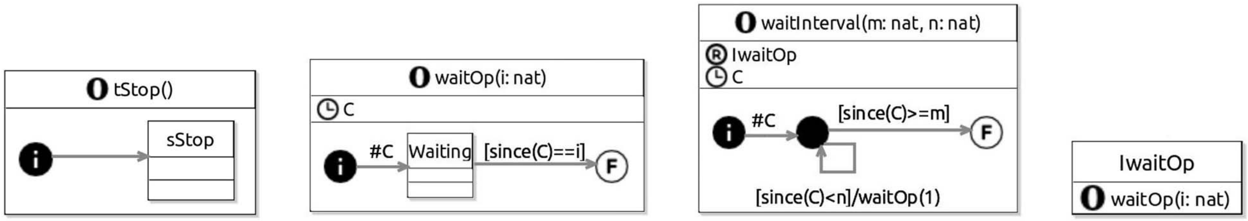


Figure 11. Normalized basic operations.

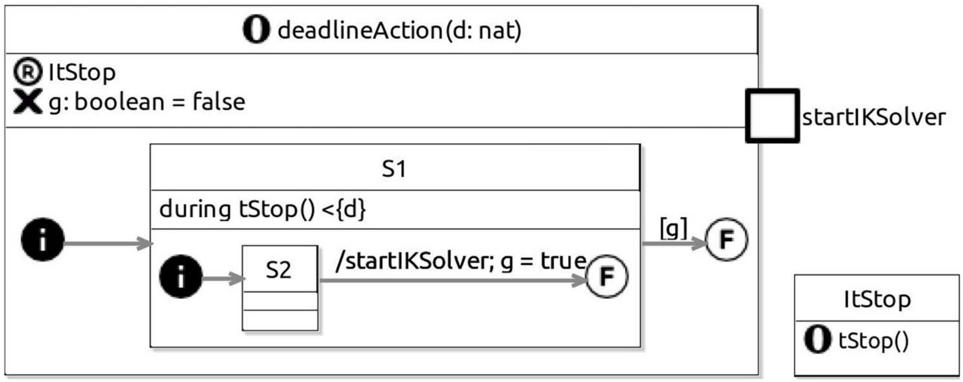


Figure 12. Example: deadline—not normalized.

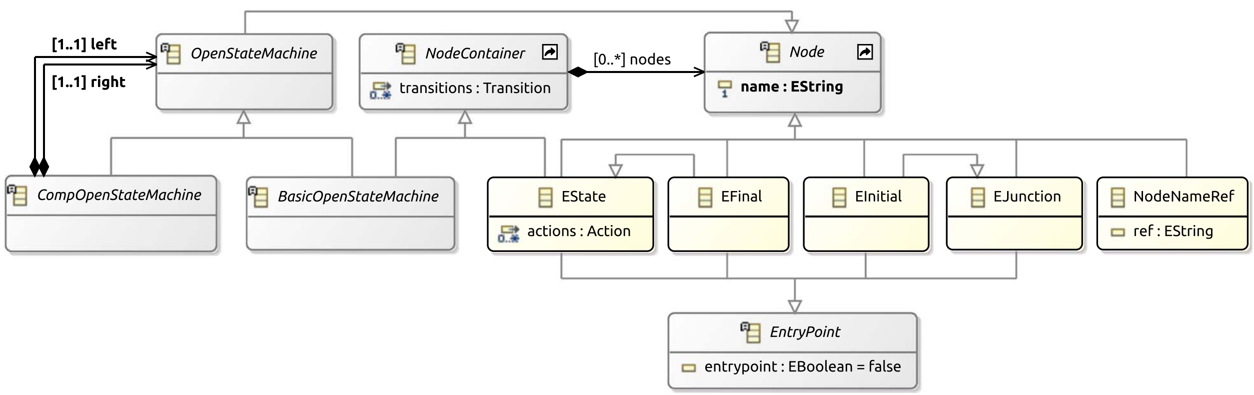


Figure 13. Metamodel for open machines.

are similar to the State, Final, Junction and Initial classes of the RoboChart metamodel (Fig. 5), but inherit from a new class EntryPoint, with the extra attribute.

In addition, all Nodes have a name as an attribute. This includes the Junctions and Final states, since they can all be the target of a transition via a NodeNameRef, which uses a name to identify a Node. (A modelling tool can easily generate names for Nodes, other than States, and hide those not relevant, to avoid burdening the modellers and cluttering the models.)

Since an OpenStateMachine is a Node, a composite state can use the new composition operator to define its machine. So, we can take advantage of the new operator for stepwise compositional transformation of machines at all levels. Well-formedness conditions ensure that a composite state either includes one OpenStateMachine with exactly one InitialState, or one or more nodes of other types, but not both, and that OpenStateMachines are not the target or source of transitions.

The notion of normalization for OpenStateMachines in general is standard, in that it requires them to be a composition of normalized machines. For a BasicOpenStateMachine, we have the conditions already presented for NodeContainers, and an extra condition.

Definition 3.3. [Normal form for OpenStateMachine] An OpenStateMachine is normalized if it is a normalized BasicOpenStateMachine, or a CompOpenStateMachine whose left and right OpenStateMachines are normalized. A BasicOpenStateMachine is normalized if it is a normalized NodeContainer, according to NCNF1-2 (Fig. 7) and BOMNF1-2 in Fig. 14.
With BOMNF1, we ensure that there is at most one non-connecting node in a normalized BasicOpenStateMachine. So, every transition is either a self-transition or a transition to a NodeNameRef. With BOMNF2, we ensure that any machines in composite states are also normalized, potentially as a CompOpenStateMachine. Together, these conditions ensure that the control flow of a normalized OpenStateMachine is fully exposed using the state machine combinator ⊙.

BOMNF1 The subset of nodes that are not NodeNameRefs is either empty or a singleton.
BOMNF2 All its OpenMachines are normalised.

Figure 14. Normal form for a BasicOpenStateMachine—a normalized BasicOpenStateMachine satisfies NCNF1, NCNF2 and BOMNF1 and BOMNF2 here.

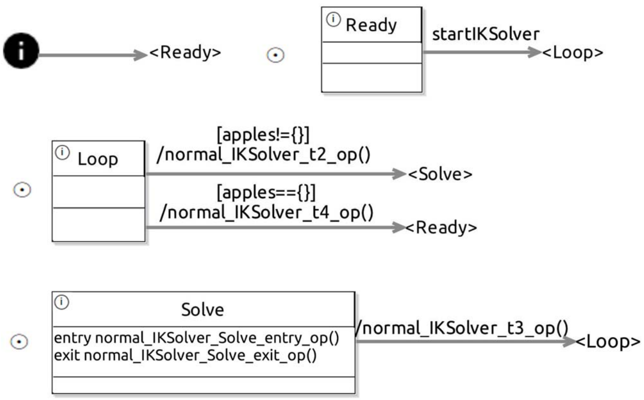


Figure 15. IK solver—Open Machine normalized.

Fig. 15 shows a normalized version of the open machine for the IKSolver on the top of Fig. 4. In Fig. 15, to ensure that every basic open machine has at most one node that is not a reference to a node defined in another machine, decompositions using the ⊙ operator split the initial state, and the states Ready, Loop and Solve into different machines. In addition, the statements in all actions are operation calls.

Next, we define how we can normalize an arbitrary machine via the systematic application of three sets of algebraic laws for StateMachineDefinitions, OperationDefinitions and OpenStateMachines.

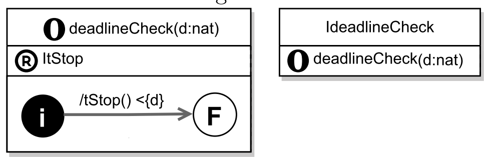
4. NORMALIZATION AND LAWS

We structure the presentation of our laws by describing their role in normalization strategies for RoboChart and open machines. In Section 4.1 we present the procedure for StateMachineDefinitions; in Section 4.2, we have the procedure for OperationDefinitions; and, finally, in Section 4.3, we have the procedure for open machines. In each case, we show that the procedure terminates and produces a normalized machine, thus establishing a relative notion of completeness for our sets of laws.

4.1. normalization: StateMachineDef

We first consider a procedure normSMB() to normalize a RoboChart StateMachineDefinition according to Definition 3.1. This procedure is shown in Fig. 16. (It applies more widely to any StateMachineBody, but we focus here on normalization of a StateMachineDefinition. In Section 4.2, we explain that normSMB() is useful, although not enough, to normalize an OperationDefinition too.) The approach is first to eliminate all sinceEntry expressions (Step 1), introduce operations that execute each of the Statements, and associated interfaces that declare those operations, and finally use them to replace all Statements with operation Calls (Step 2). Next, we eliminate the deadlines in transitions: we again declare new operations and interfaces (in Steps 3 and 4), later used (in Step 5) to encode the deadlines using Calls.

1. Apply Law 1 exhaustively.
2. Apply Laws 2 and 3 exhaustively whenever s is not a Call.
3. Apply Law 4 with argument tStop() in Figure 11, and Law 5 with argument ItStop in Figure 12.
4. Apply Laws 4 and 5 with the operation and interface below as arguments.



5. Apply exhaustively Law 6.

Figure 16. normSMB()—normalization of StateMachineBody.

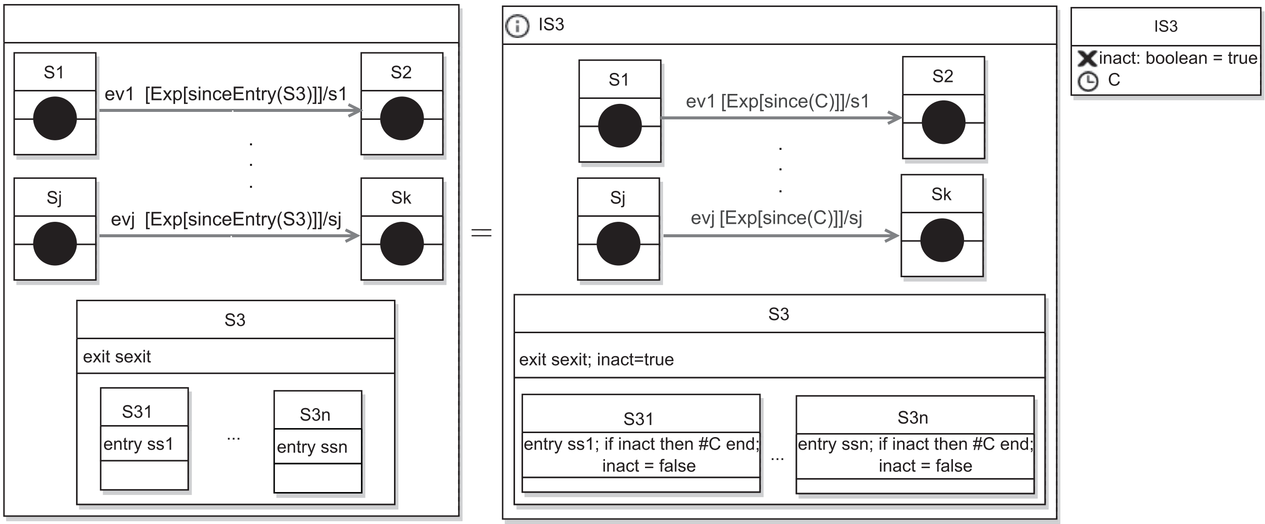
Below, we describe each of the steps. For each of them, we present and describe the laws that are needed. In a law definition, we name it, identify its arguments and use a(n informal) diagrammatic notation to describe an equality. All laws, however, are also stated formally as described in Section 6. Law definitions may also include a **provided** clause that imposes restrictions that must be satisfied for a law application to be valid. Finally, a law definition can include a **where** clause that defines elements in the body of the law (equality and **provided** clause) that are not given as argument or defined by pattern matching. In our normalization procedures we apply all laws from left to write, although they are all equalities.

Step 1 Here, for each state S3 for which there exists one or more sinceEntry(S3) expressions (necessarily in transition guards, which is the only context where such expressions can occur), we apply Law 1 once.

Law 1 establishes that we can replace all occurrences of sinceEntry(S3) expressions with since(C), where C is a new clock that is initialized once S3 is entered. On the left-hand side of the equality in Law 1, the block named S3, inside the unnamed block, stands for any State in a StateMachineBody. As stated in the proviso, the blocks named S31, ..., S3n are those identified by the function innermostInitialStates(S3). They are the innermost states of S3, if any, that can be reached by a sequence of transitions, starting from the transition from the initial junction of the machine of S3, and including only transitions from initial junctions of composite states or from junctions. If S3 is not a composite state, this set of states is empty. If S3 is composite, and its state machine has a transition from the initial junction to another junction, and from there to two non-composite states S31 and S32, for example, then this set includes S31 and S32. (For compatibility with UML, there can be only one transition from the initial junction.) If S31 or S32 is composite, then, instead of including it, we consider the initial junction of its machine, and so on. So, the innermost states so defined are not composite. The exit action sexit of S3, and the entry actions ss1,..., ssn of S31,..., S3n, if any, are explicitly indicated. Finally, the transitions where sinceEntry(S3) occurs have arbitrary triggers ev1,..., evj and actions s1,..., sj. The state blocks named S1,..., Sj, S2,..., Sk, with a junction symbol inside, represent Nodes: States or Junctions. The transitions identified in Law 1 can be between any Nodes, including those inside S3, if any.

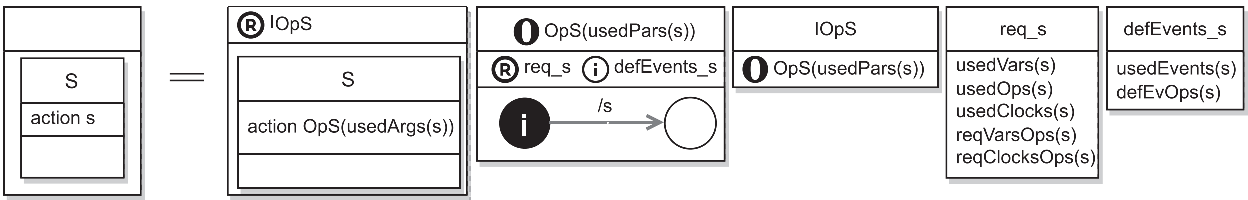
On the right-hand side of Law 1, a new interface IS3 declares a new boolean variable inact, which is initialized to true, and the new clock C. This interface is declared as defined in the StateMachineBody, so that inact and C are added to its local context. The clock is reset (#C) after the entry actions ss1,..., ssn of the inner

Law 1. elim-sinceEntry



provided The names IS3, inact, and C are fresh, and innermostInitialStates(S3) = { S31, ..., S3n }

Law 2. intro-call-for-act-state()



provided The names OpS, IOpS, req_s, and defEvents_s are fresh.

states S31, ..., S3n, since it is after executing one of these actions that entering S3 is concluded. This clock reset, however, should occur just the first time these states are entered, as a result of the machine entering S3. For example, if S31 has as self-transition, or the control flow leads back to S31 in any other way, without having left S3, then C should not be reset because C is used to record the time since the last entry in S3. For this reason, we use inact to flag whether it is the first time an inner state has been entered. At the end of entry actions of S31,..., S3n, inact is set to false, and then set back to true only when S3 is exited: after its exit action sexit.

The proviso of Law 1 requires the names of the new interface, variable and clock to be fresh in the model. The particular names used are not important, and soundness is guaranteed as long as they are fresh and used consistently as determined in the law definition.

Law 1 captures the meaning of sinceEntry(S) expressions. It also highlights the convenience of the availability of these expressions, since control based on clocks can become convoluted.

Step 2 Here, for each Statement that defines an action in a State or Transition, and is not already a Call, we apply Law 2 or 3. These laws are applied, exhaustively, that is, until all Statements are a Call.

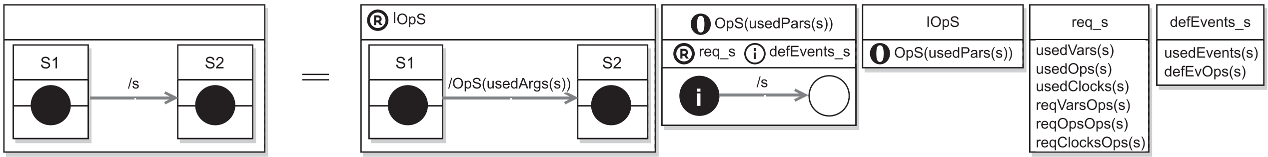
Law 2 establishes that any Statement s in any action of any state S of a StateMachineBody can be replaced with a Call to an action operation OpS for s. On the left-hand side of the equality, action s stands for an EntryAction, DuringAction or ExitAction with Statement s. On the right-hand side, OpS(usedPars(s)) and its associated interface IOpS are defined, the StateMachineBody declares IOpS as a required interface, and the Statement s in the state S identified on the left-hand side is replaced with a Call to OpS(usedArgs(s)). Finally, on the right-hand side, the interfaces req_s and defEvent_s used in the definition of OpS are declared.

The proviso of Law 2 requires the names of the new operation and interfaces to be fresh. Like in Law 1, the particular names used are not important, and soundness is guaranteed if they are fresh and used as indicated.

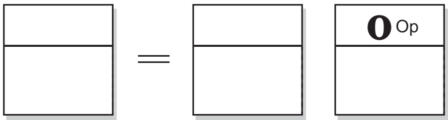
As noted, Law 2 and normSMB() are applicable to any StateMachineBody, not only a StateMachineDefinition. If Law 2 is applied to an OperationDefinition, as opposed to a StateMachineDefinition, the Statement s may use parameters of the operation. With the application **usedArgs**(s) of a simple syntactic function, we identify the list of names of these parameters, which are passed as arguments in the call to OpS. Moreover, with **usedPars**(s), we determine the matching declarations of the parameters. These are used to specify the signature of OpS in its definition. If there is no use of parameters in s, **usedArgs**(s) is the empty list of arguments, and, of course, **usedPars**(s) is the empty list of declarations. When Law 2 is applied to a StateMachineDefinition, which has no parameters, it is certain that **usedArgs**(s) and **usedPars**(s) are empty lists.

In the interfaces req_s and defEvent_s, we declare all the variables, operations, clocks and events used in the Statement s. Using

Law 3. intro-call-for-act-transition()

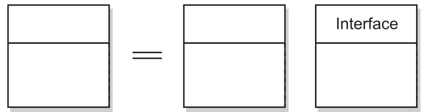


Law 4. intro-op(Op)



provided id(Op) is a fresh name.

Law 5. intro-interface(Interface)



the applications usedVars(s), usedOps(s), usedClocks(s) and usedEvents(s) of additional syntactic functions, we determine the declarations of the variables, operations, clocks and events directly referenced in s. Via reqVarsOps(s) and reqClocksOps(s), we get the declarations of the variables and clocks in the definitions of the operations in usedOps(s). These variables and clocks can be declared in interfaces or directly in the definition of an operation in usedOps(s). Similarly, with defEvOps(s), we get the declarations of events in the operations in usedOps(s). Since all these functions may identify no declarations, or rather empty sets of declarations, the new interfaces req˙s and defEvents˙s may be empty. Additionally, in req˙s and defEvent˙s, we get the union of the sets of declarations identified by the function applications. So, no repeated declarations are included in the interfaces.

The definition of OpS requires req˙s and defines defEvents˙s. Since events are points of interaction, they cannot be required, but are always defined.

For the machine IKSolver in Fig. 3, after the exhaustive application of Law 2, we have declarations for operations normal˙IKSolver˙entry˙op() and normal˙IKSolver˙exit˙op(). The entry and exit actions of the state Solve in IKSolver are replaced with calls to these operations as shown in Fig. 8.

Soundness of Law 2 is discussed in Section 6.2.1.

Law 3 is similar to Law 2, but considers all actions s in Transitions. After the exhaustive application of Law 3 to IKSolver, the operation definition in Fig. 10 as well as definitions for normal˙IKSolver˙t3˙op() and normal˙IKSolver˙t4˙op(), and associated interfaces, such as, req˙IKSolver˙t2 in Fig. 10, are introduced in the model. In addition, the definition of IKSolver is transformed to that shown in Fig. 8.

Steps 3 and 4 Here, we introduce in the model the definitions of the operations tStop() in Fig. 11 and deadlineCheck(d:nat) in Fig. 16, and the associated interfaces ItStop in Fig. 12 and IdeadlineCheck also in Fig. 16. These definitions are used in the following step to eliminate deadlines from transitions. We apply Laws 4 and 5 to introduce these definitions.

Law 4, named intro-op, establishes that we can always introduce a new operation in a model. Accordingly, its proviso requires the name id(Op) of the OperationDefinition Op given as argument to be fresh. In the body of Law 4, the left-hand side of the equality has an unnamed block, which, as already explained, stands for an arbitrary StateMachineBody. On the right-hand side of the equality, we extend the model with Op. This is indicated by repeating the unnamed block and including a block labelled Op.

Law 5, which is is similar to Law 4, establishes that we can declare a new interface in a model.

For simplicity, we assume that the names of the operations and interfaces defined in Steps 3 and 4 are fresh. If this is not the case, different fresh names need to be used. The particular names adopted have no bearing in the soundness of the strategy.

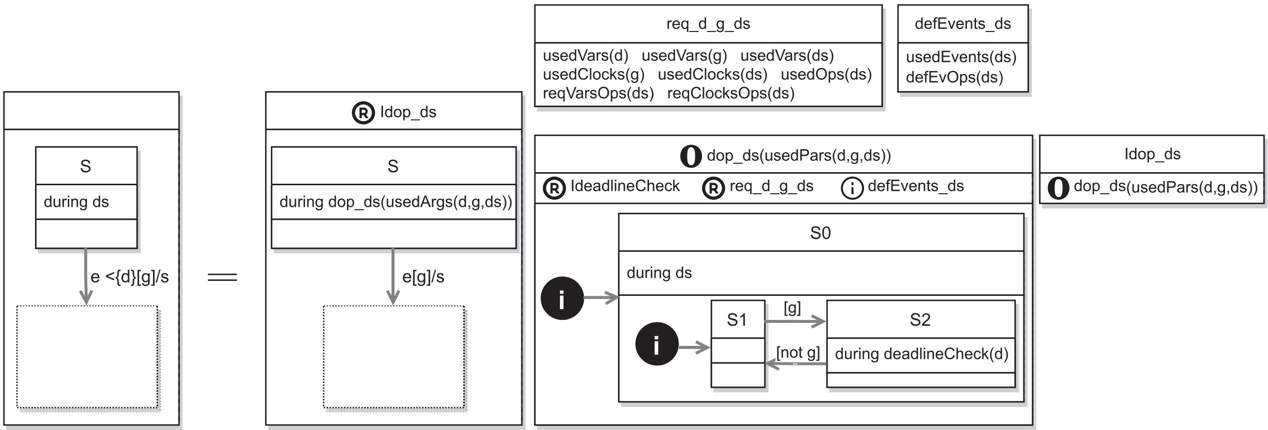
Step 5 Here, we apply Law 6 exhaustively to remove the deadlines in transitions by replacing, or introducing, during actions in the source states S of these transitions with calls to operations dop˙ds. To eliminate a deadline from a transition, Law 6 uses an operation dop˙ds to enforce that deadline using a machine in a composite state S0 defined in dop˙ds. If S has a during action, its statement ds is run in parallel with that machine, as required: it becomes a during action of the state S0.

The deadline d of a transition becomes relevant only once its guard holds. So, the machine of S0 remains in a state S1 until the guard g of the original transition holds. At that point, it moves to the state S2, where a call deadlineCheck(d) enforces the deadline d. In deadlineCheck(d) (see Fig. 16, Step 4), a call tStop() blocks, but has a deadline d to terminate. Since tStop() does not terminate, for this deadline to be met, dop˙ds() has to be interrupted. This can be achieved only by S exiting, when its during action dop˙ds() is interrupted. For that, the transition labelled e[g], or some other transition out of S, must take place. In either case, the original deadline d on the transition is enforced.

Law 6 captures precisely the semantics of a deadline on a transition: it is not a deadline on its trigger. First, the deadline is relevant only if the guard holds. Secondly, if some other transition is taken, the deadline is cancelled. What we actually have is a deadline on exiting the source state of the transition. Soundness of Law 6 is the topic of Section 6.2.2.

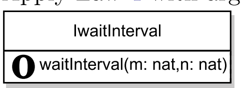
To summarize, first of all, normSMB() terminates. The potential sources of non-termination are the exhaustive law applications in Steps 1, 2 and 5. In each iteration of the Steps 1 and 5, however, Laws 1 and 6 are applied from left to right, eliminating the patterns to which they apply. So, the steps terminate when all the instances of these patterns are eliminated. In Step 2, termination is ensured by the restriction that Law 2 is applied only for statements s that are not a Call, and such statements are replaced with Call statements. So, when all such statements are eliminated, Step 2 terminates. Secondly, following Step 2, NCNF1 holds. Moreover, after Steps 3 and 4, Law 6 applies to every transition with a deadline. So, after Steps 1 and 5, NCNF2 holds. Overall, NCNF1 and NCNF2 both hold and the StateMachineDef is normalized as required.

Law 6. elim-deadline-transition()



provided IdeadlineCheck and deadlineCheck as defined in Step 4 of Figure 16 are in scope.

1. Apply Law 4 with argument tStop() (see Figure 11), and Law 5 with argument ItStop (see Figure 12).
2. Apply Law 4 with argument deadlineCheck(d:nat), and Law 5 with argument IdeadlineCheck (see Step 4 in Figure 16).
3. Apply Law 4 with argument waitOp(i:nat), and Law 5 with argument IwaitOp (see Figure 11).
4. Apply Law 4 with argument waitInterval(m:nat,n:nat) (see Figure 11), and Law 5 to the interface below.



5. Apply exhaustively Laws 7-13.
6. If the OperationDefinition is not an action operation, then apply exhaustively Law 3 whenever s is not a Call.

Figure 17. normAO()—Procedure for normalization of RoboChart action OperationDefinitions.

4.2. normalization: OperationDef

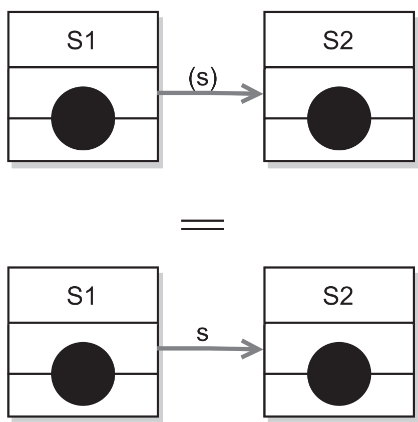
To normalize an OperationDefinition, we consider two cases. If it is not an action operation, we apply the procedure normSMB() (Fig. 16). (As said, normSMB() applies to any StateMachineBody, not only to StateMachineDefinitions.) Using normSMB(), we remove the structure from the actions and deadlines in the OperationDefinition out to (further) OperationDefinitions, just like it is done for StateMachineDefinitions.

Secondly, if the OperationDefinition is an action operation, we use the procedure normAO() in Fig. 17. It starts by introducing, in Steps 1–4, the support operations already presented for use in normalization of time behaviour, and their associated interfaces. Afterwards, we apply, in any order, exhaustively, Laws 7–13 to flatten the structure of the action (Step 5). In doing so, we may introduce additional junctions, so that the resulting OperationDefinition may no longer be an action operation. In this case, in Step 6, we introduce operations for the actions, like in Step 1 of normSMB().

Law 7 is simple: it eliminates spurious parentheses around Statements that define transition actions. Laws 8 to 10 relate structure in a transition action with that in the StateMachineBody as a whole. Law 8 equates a sequence s1; s2 of actions to a sequence of transitions with actions s1 and s2 connected by a junction.

Law 9 describes how a conditional if b then s1 else s2 end in a transition can be encoded by a pair of sequences of transitions, where b and not b are guards for the first transitions in the pairs. The subsequent transitions have the then and else Statements as actions. Since, as already said, for compatibility with UML, there can be just one transition out of an initial junction, a proviso

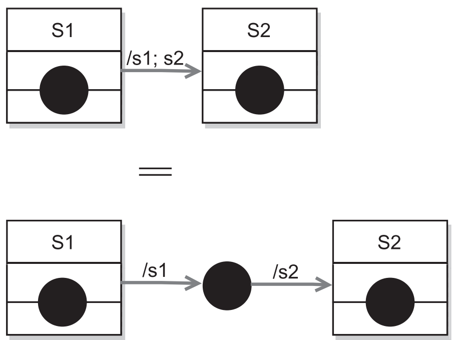
Law 7. elim-parentheses()



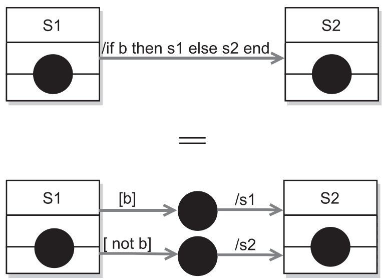
ensures that Law 9 is not applicable if the source S1 of the transition is an initial junction. To deal with such transitions, an additional Law 10 permits the introduction of an intermediate junction between the initial junction and the target Node. After an application of Law 10, Law 9 applies to the intermediate junction and its outgoing transition with action s.

Law 11 allows the elimination of a deadline on a transition action. It is in many ways similar to Law 6. Law 11, however, applies to a deadline on a transition action, as opposed to a deadline on a transition itself. Despite that, Law 11 considers a transition action s<{d} and, like Law 6 defines and calls an operation

Law 8. split-sequence()

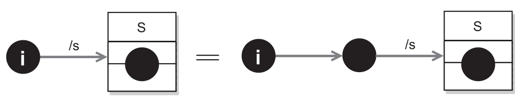


Law 9. split-conditional()



provided S1 is not an Initial junction.

Law 10. intro-junction-from-initial-junction()



(deadlineAction˙s(d:nat,usedArgs(s))) to enforce the deadline in place of the original s<{d}.

The definition of deadlineAction˙s is slightly simpler than that of dop˙s in Law 6 because it does not deal with a guard (or trigger) in the transition. In normAO(), Law 11 is applied to machines arising from the application of Steps 1–4 and Laws 7–13 to an action operation. Since an action operation does not have a guard or a trigger in its only transition, and they are not added by any of the Steps 1–4 and Laws 7–13, Law 11 is only applied to transitions without guards or triggers.

The operation deadlineAction˙s(d:nat,usedPars(s)) executes s in parallel with deadlineCheck(d), via the definition of a composite state S0 with deadlineCheck(d) as a during action and with s in an action of a transition from a substate S1 of S0 to a final state. To satisfy the original deadline, s must be executed, and finished, before more than d time units have passed. To ensure execution of s, exit from S0, and so termination of deadlineAction˙s, is predicated on a transition with a guard g being enabled. This is possible only after s is executed, since g is initialized to false, and assigned true only after s finishes. To ensure that no more than d time units are passed, the operation deadlineCheck(d) is used. We recall that it enforces the deadline on tStop(), and that can be met only when S0 is exited so that deadlineCheck(d) is interrupted. In this way, the deadline is indirectly enforced on s, because exit from S0 is predicated on its outgoing transition being enabled.

The proviso of Law 11 requires that the operation deadlineCheck(d:nat,usedPars(s)) and an interface that declares it are in scope. In normAO(), this is ensured by Step 2. So, deadlines in actions are all eliminated.

Finally, we have Laws 12 and 13 to eliminate Wait Statements in favour of use of the operations in Fig. 11 already explained. They use a clock to capture the timed behaviour of Wait Statements.

Laws 7–13 are applied in our strategy just to action operations, or to machines arising from the application of these laws to action operations. As a consequence, we have not considered in these laws the possibility that the transitions have guards or triggers. The generalization of these laws is, however, straightforward. In all cases, except in Laws 8 and 9, the guard and trigger, if any, do not need to be changed. In the case of Law 8, they are to be part of the label of the transition to the new junction, and in Law 9, they need to be duplicated in the transitions to the new junctions.

After applications of Laws 8–10, the OperationDefinition is no longer an action operation. In this case, Step 6 ensures that it satisfies OPDNF2 by exhaustive application of Law 3. If, however, Laws 8–10 have not been used, the result is still an action operation.

In summary, if we apply normSMB(), as established in the previous section, the result is an operation definition that satisfies OPDNF2, which corresponds to the normal form definition for StateMachineDefinitions. If we use normAO() termination is guaranteed because, in Step 5, each law eliminates the pattern to which it applies, and does not introduce it or the pattern relevant for any of the other laws, with one exception. As said, Law 10 potentially introduces a pattern to which Law 9 might apply, but Law 10 applies at most once for each initial junction. Moreover, in Step 6, termination is guaranteed by the proviso that the statement s is not a Call. (This step is similar to Step 2 of normSMB().) Finally, we note that with the exhaustive application of Laws 7 and 11–13, we enforce OPDNF1. In detail, the forms of Statement disallowed by OPDNF1 are eliminated as follows.

- ParStmt is a parenthesized Statement, which is eliminated by Law 7.
- Sequences (SeqStatement) and conditionals (IfStmt) are not present, since Laws 8 and 9 have been applied exhaustively.
- TimedStatement is a deadline, eliminated by Law 11 (using additional operations).
- Wait is a wait, eliminated by Law 12 or 13, depending on whether it is nondeterministic or not.

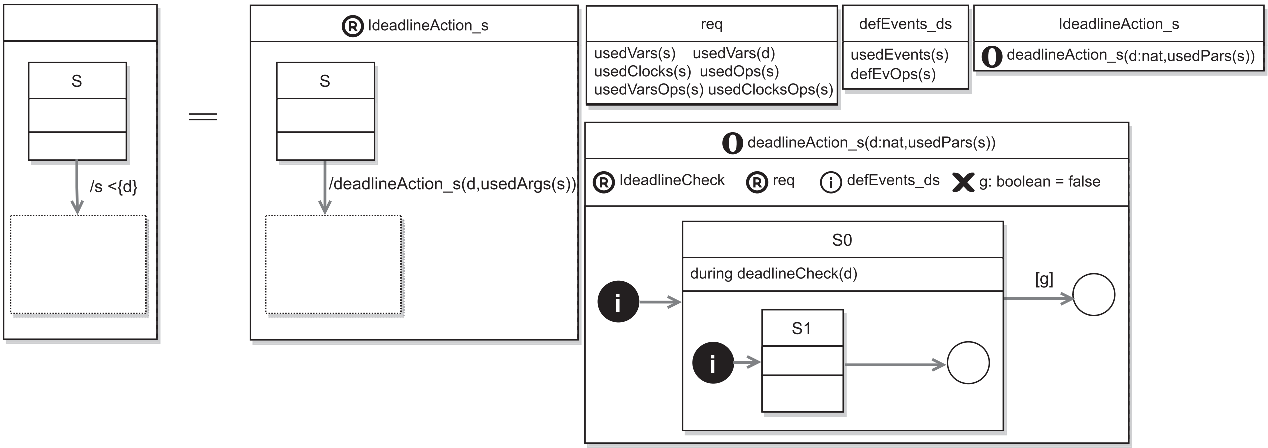
Overall, the resulting OperationDefinition is normalized according to the normal form in Definition 3.2.

4.3. Normalization: open machines

To normalize an OpenStateMachine, we use the procedure normOM(openM) in Fig. 18 that takes a machine openM as argument. If openM is a BasicOpenStateMachine, then in Step 1a we use a procedure normNC() for NodeContainers. This procedure is very similar to normSMB() from Fig. 16; it is presented in Fig. 19. The differences are related to the Laws 1, 2, 3 and 6 used in normSMB() that apply to a StateMachineBody and enrich its Context with an additional interface. For normNC(), we need similar laws that, however, apply to a NodeContainer, without a Context. In addition, in normNC() there is no need to declare operations and interfaces for later use. That suitable operations are used is ensured by the provisos of the new laws as detailed in the sequel.

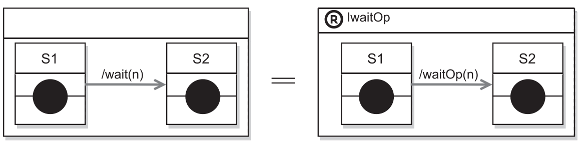
After an application of normNC(), all sinceEntry(S) expressions are eliminated, all actions become operation Calls, and all deadlines are eliminated. Afterwards, in Step 1b of normOM(openM),

Law 11. elim-deadline-action()



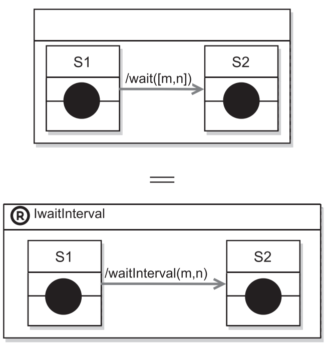
provided IdeadlineCheck and deadlineCheck as defined in Step 4 of Figure 16 are in scope.

Law 12. elim-wait()



provided IwaitOp and waitOp as defined in Figure 11 are in scope.

Law 13. elim-nondeterministic-wait()



provided IwaitInterval as defined in Figure 11 and waitInterval in Step 4 of Figure 17 are in scope.

1. If openM is a BasicOpenStateMachine, then
   (a) Apply normNC()
   (b) Apply decompBOM exhaustively
2. If the given machine openM is a CompOpenStateMachine, then execute normOM(openM.left) and normOM(openM.right).

Figure 18. normOM(openM)—procedure for normalization of an open machine.

we apply the procedure decompBOM(openM) in Fig. 20 to decompose the machine using the combinator ⊙. This procedure is applied exhaustively, so that both the machine itself and any machines in its composite states are decomposed. In Step 2, we consider CompOpenStateMachines, already defined using ⊙, and apply normOM recursively to the composed machines (openM.left and openM.right).

1. Apply Law 14 exhaustively.
2. Apply Laws 15 and 16 exhaustively whenever s is not a Call.
3. Apply exhaustively Law 17.

Figure 19. normNC()—normalization of NodeContainer.

1. If openM has an EntryPoint EP with a transition whose target is not itself an EntryPoint and not a NodeNameRef, then
   (a) Apply Law 18
   (b) Apply decompBOM(right), where right is the rightmost machine generated in Step (a).
2. elseif there is more than one EntryPoint, then
   (a) Apply Law 18

Figure 20. decompBOM(openM)—procedure for decomposition of a BasicOpenStateMachine.

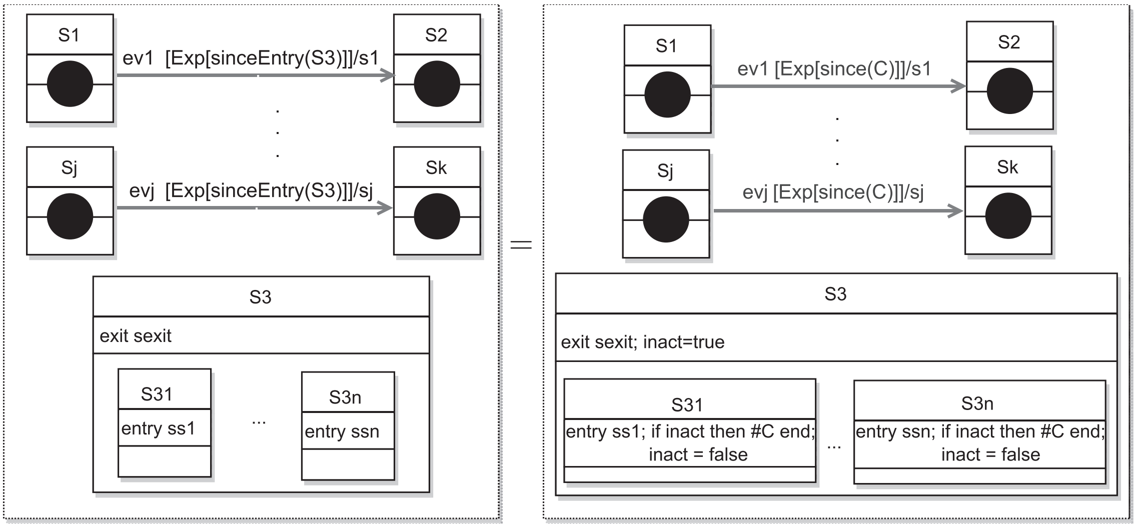
Next, we describe in detail the steps of the procedures normNC() and decompBOM(openM).

Procedure normNC() As shown in Fig. 19, in Step 1 of normNC(), instead of Law 1, we apply Law 14. It differs just in that it does not declare the interface IS3, the variable inact or the clock C to enrich the context of the machine. Since it is an OpenStateMachine, and so a NodeContainer but not a Context, there is no context to be enriched. The unnamed dotted boxes in the body of Law 14 denote OpenStateMachines. In these, variables and clocks can be used without declaration, and the proviso ensures that they are new.

Similarly, in Step 2 of normNC(), instead of Law 2, we apply Law 15, different in that it does not introduce a new operation OpS that executes the Statement s of a state S. Every element used in s is already in scope and the proviso requires that OpS is defined as indicated.

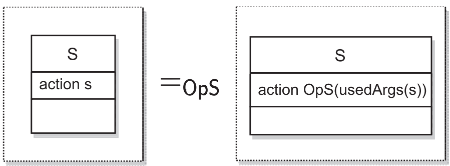
The notion of equality $OM_1 =_{OpDS} OM_2$ for the laws of open machines is parametrized by a set of operation definitions $OpDS$. In Law 14, this set is empty and omitted. In Law 15, the argument is the singleton set {OpS}, briefly indicated as just OpS. We recall

Law 14. elim-sinceEntry-nc()



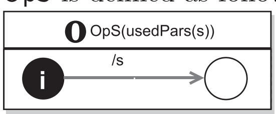
provided The names inact and C are fresh, and innermostInitialStates(S3) = { S31, ..., S3n }

Law 15. intro-call-for-act-state-nc()

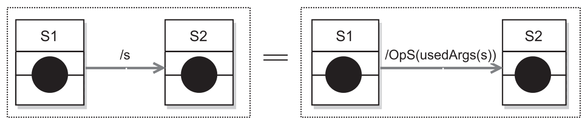


provided

- The name OpS is fresh.
- OpS is defined as follows.

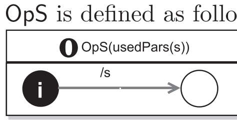


Law 16. intro-call-for-act-transition-nc()



provided

- The name OpS is fresh.
- OpS is defined as follows.



that, as illustrated in Law 15, an operation definition identifies the name of the operation, its parameters and its behaviour (using, for instance, a state machine). As formalized in Section 6.2, with $OM_1 =_{Op} OM_2$ we establish that OM_1 and OM_2 are equal provided, when an operation whose name is that of Op is called, then OM_1 and OM_2 both call Op itself. In an open machine, the operation that is called using a given name is not identified in the machine. Such an open machine can be used in the scope of various definitions for the named operation. The equality $OM_1 =_{Op} OM_2$, therefore, fixes the association of Op to its name. More generally, the equality $OM_1 =_{OpDS} OM_2$ similarly fixes the association of all operation definitions on $OpDS$ to their names. In Law 15, the operation OpS is not called in the open machine on the left-hand side, since the proviso requires that the name OpS is fresh.

In the procedure normNC(), law applications with different arguments are used: one operation definition for each action in a state or transition that is not a Call, for example. Overall, normNC() establishes equality parametrized by the set of all operation definitions used as arguments. This follows from the fact that $OM_1 =_{Op1} OM_2$ and $OM_2 =_{Op2} OM_3$ imply that $OM_1 =_{\{Op1,Op2\}} OM_3$. Generally, $OM_1 =_{SO1} OM_2$ and $OM_2 =_{SO2} OM_3$ imply $OM_1 =_{SO1 \cup SO2} OM_3$.

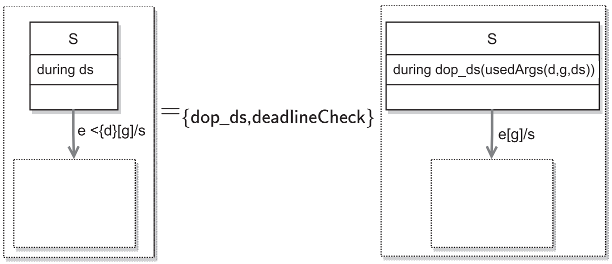
Laws 16 and 17 are similar to Laws 3 and 6, but, like Law 15, do not introduce operations and interfaces.

In Steps 3 and 4 of normSMB(), we introduce operation definitions and interfaces for use in the last step. These are not needed in normNC(). Instead, we have a step similar to Step 5, which applies Law 17. It is similar to Law 6, and establishes equality with the definitions for dop_s and deadlineCheck as arguments.

Procedure decompBOM(openM) The core of this procedure is the application of Law 18, which splits a BasicOpenStateMachine by isolating its entry points in separate machines combined by ⊙. For example, in a traditional machine (such as a UML machine), the single EntryPoint is the initial junction, with a single transition from that EntryPoint to a different Node S. In this case, applying Law 18, we obtain the ⊙ composition of two BasicOpenStateMachines. The first has the initial state with a transition to a NodeNameRef to S. The second has all states and transitions of the original machine, except the initial junction and its transition; in this machine, S becomes an EntryPoint. Fig. 21 shows the result for the IK solver in Fig. 3, taken as an OpenMachine, after an application of Law 18.

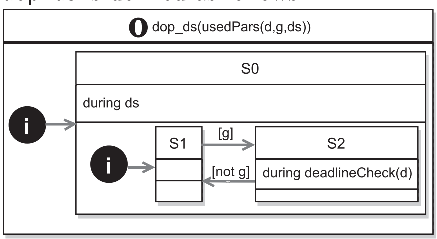
In general, Law 18 applies to a BasicOpenStateMachine with any number of EntryPoints, defining a BasicOpenStateMachine for

Law 17. elim-deadline-transition-nc()

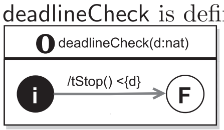


provided

- The names dop_ds and deadlineCheck are fresh.
- dop_ds is defined as follows.



- deadlineCheck is defined as follows.



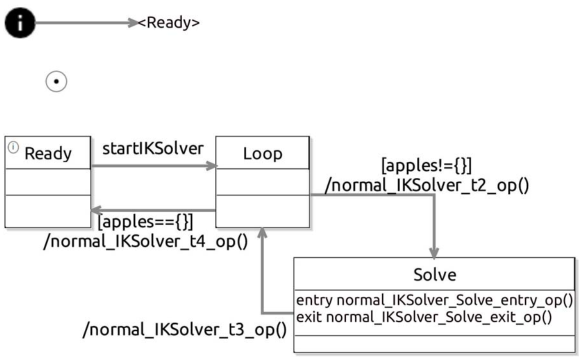


Figure 21. IK solver—Open Machine—Step 1a—First iteration of decompBOM(IKSolver).

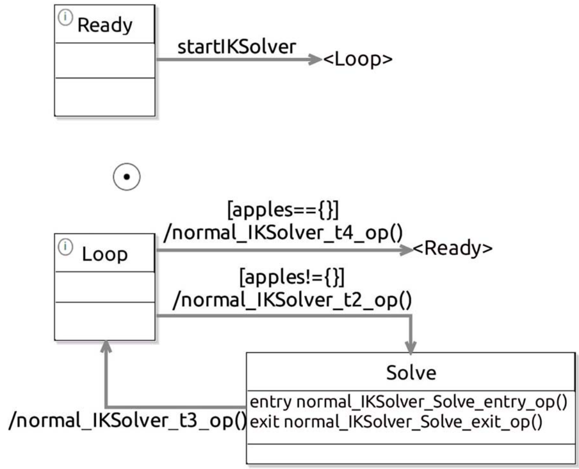


Figure 22. IK solver—Open Machine—Second iteration of decompBOM(IKSolver).

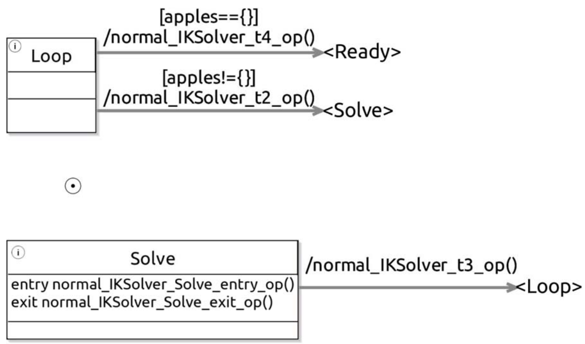


Figure 23. IK solver - Open Machine - Third iteration of decompBOM(IKSolver).

each of them. The equality it establishes has the empty set of operation definitions as argument, which is omitted for simplicity. From each EntryPoint S1,...,Sn, there may be self-transitions, or transitions to a NodeNameRef (indicated by <S11>,..., <Sn1>) or to any other form of Node (S1m,...,Snm), that is, Junction or (Final) State. The first two forms of transition are untouched by Law 18; the latter are replaced with new NodeNameRefs to the original Nodes.

A final BasicOpenStateMachine includes (a) all the actual Nodes S1m,...,Snm, which become EntryPoints; (b) the sets of all other Nodes; (c) Transitions from the original machine that do not target one of the original EntryPoints S1,...,Sn; (d) Transitions that target those original EntryPoints S1,...,Sn, with a new NodeNameRef for those EntryPoints as a target; (e) the new NodeNameRefs as needed for (d). This replacement of the targets of the transitions is indicated in Law 18 using the substitution [S1,...,Sn <S1>,...,<Sn>].

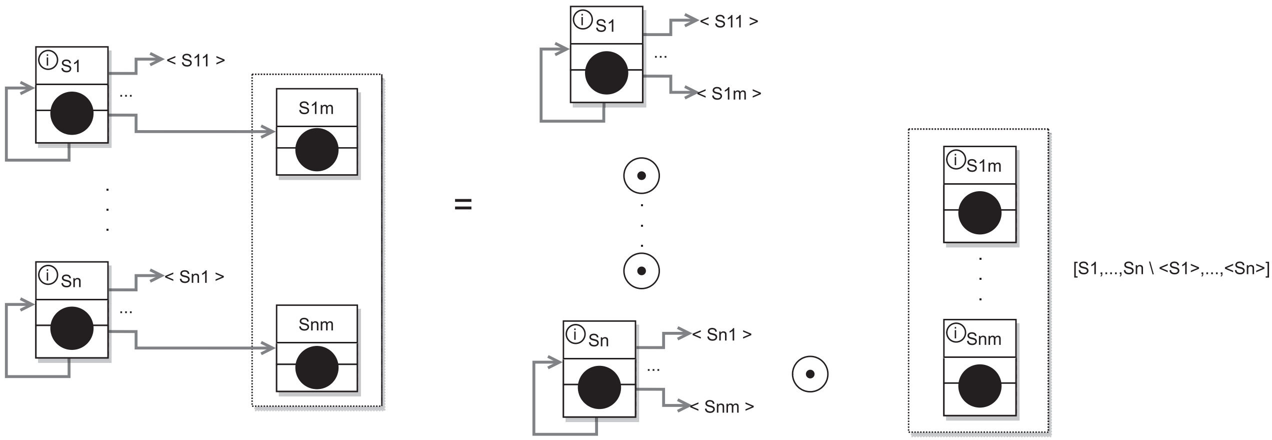
Figure 22 shows the result of applying Law 18 to the right machine in Fig. 21. Because it has only one EntryPoint, the decomposition gives rise to just two BasicOpenStateMachines. The first is normalized. The second needs to be further decomposed via additional applications of Law 18. There, the transition from Loop back to Ready, which is an EntryPoint, is now a transition to a NodeNameRef that names Ready instead.

The normalization procedure decompBOM(openM) for a BasicOpenStateMachine openM applies Law 18 whenever openM has at least one EntryPoint with a transition t to a different Node that is not an EntryPoint or a NodeNameRef (Step 1), or, if not, when it has multiple EntryPoints (Step 2). In the first case, openM has more than one node that is not a NodeNameRef: at least the EntryPoints and the target of the transition t. This violates BOMNF1, so in Step 1a, Law 18 is applied to split openM. Since the rightmost machine right that results from applying Law 18 needs to be considered, in Step 1b a recursion decompBOM(right) deals with it.

In Fig. 22, the right machine has the state Loop as an EntryPoint with a transition to the State Solve, which is not an EntryPoint. So, with the recursive call to decompBOM, we apply Law 18 to that machine to obtain the result in Fig. 23. Since in the right machine the only transition is to a NodeNameRef, and that machine has a single EntryPoint, no more changes arise from a recursive call to decompBOM.

In Step 2, we consider the case where there are multiple EntryPoints, but no transitions to a Node that is not an EntryPoint or a NodeNameRef. In this case, we apply Law 18 in Step 2a, but there is no need for a recursive call. Since there are no Nodes besides the EntryPoints, the rightmost machine resulting from applying Law 18 is empty and omitted.

Law 18. comp-from-entry-points()



To summarize, the procedure normOM(openM) terminates and normalizes any OpenStateMachines. If openM is a BasicOpenStateMachine, it terminates because normNC() and decompBOM(openM) do: the argument for termination of normNC() is similar to that for normSMB(), and termination of decompBOM(openM) is considered next. If openM is a CompOpenStateMachine, it terminates because there is a finite number of BasicOpenStateMachines composed.

The procedure decompBOM(openM) terminates because the rightmost machine generated by Law 18, to which it is applied recursively, has a number of Nodes that are not NodeNameRefs strictly smaller than those of the original machine. This is because the EntryPoints of the original machine, which cannot be NodeNameRefs, are removed from it. In addition, no new Nodes that are not NodeNameRefs are introduced. So, since the original machine has a finite number of Nodes, after a finite number of recursive calls, there are no more Nodes that become new EntryPoints, indicating the need for further decomposition via recursive calls.

Finally, the result of normOM(openM) is a normalized OpenStateMachine. Again, we have an inductive argument. If openM is a BasicOpenStateMachine, it is normalized by normOM(openM) because normNC() ensures NCNF1 and NCNF2, and in sequence exhaustive application of decompBOM(openM) ensures BOMNF1 and BOMNF2. If openM is a CompOpenStateMachine, it is normalized because the recursive calls normalize the combined machines openM.left and openM.right.

To see that decompBOM(openM) ensures BOMNF1, we note that all machines generated by an application of Law 18, except the rightmost one, satisfy BOMNF1. For that rightmost machine, either we have a recursive call to decompBOM, so that BOMNF1 is ensured by induction, or it is empty. The empty BasicOpenStateMachine is normalized, but it is also the unit for ⊙, and can be eliminated. BOMNF2 is ensured by exhaustively applying decompBOM.

In the next section, we present examples and a tool, addressing the practical relevance of our work. We consider soundness of our laws later on in Section 6.

5. EXAMPLES AND TOOL

As said, the normal forms provide a notion of completeness for our laws. They are enough to reduce any machine to a normal form, and capture properties related to control flow and time in the state-machine notations. Here, we present in Section 5.1 the normalization of the machine AppleHarvestControl in Fig. 2, and then the normalization of a similar, but open, machine that defines the same control software. After that, in Section 5.2, we describe the tool we have developed to normalize models. It has been used with our examples and others[2] . Finally, Section 5.3 discusses and illustrates practical applications.

5.1. Normalization: Examples

In this section, we present the normalization of two examples: a RoboChart machine (Section 5.1.1) and a similar, but open, machine (Section 5.1.2).

5.1.1. *Normalization: RoboChart*

AppleHarvestControl (Fig. 2) is a StateMachineDef, and so can be normalized using the procedure normSMB() (Fig. 16). In Step 1, since AppleHarvestControl does not use sinceEntry(S) expressions, Law 1 is never applied. After Step 2, with the exhaustive application of Laws 2 and 3, the definition of AppleHarvestControl is as shown in Fig. 24. All actions are now operation Calls, and the context requires interfaces (whose names start with I_normal_) that declare these operations. Actions in (sub)states of composite states also become Calls. In [37] we find the definitions of the new operations and interfaces. No other changes are made.

In Steps 3 and 4 of normSMB(), no changes affect AppleHarvestControl directly. The laws applied in these steps enrich the scope to include definitions for the operations tStop() and deadlineCheck, with their associated interfaces ItStop and IdeadlineCheck.

In the last step, we deal with the deadline in the transition from the state Prepare to the state LocalizeFruit. With an application of Law 6, we obtain the state machine in Fig. 25. The deadline is removed from the transition, and a during action that Calls a new operation normal˙AppleHarvestControl˙Prepare˙t1˙dop() is added to Prepare. This new operation and a new interface are introduced; see [37] for their definitions.

Some, but not all, of the OperationDefinitions introduced by an application of normSMB() are already normalized. They are all action operations, and are already normalized if its single Statement satisfies the restrictions in OPDNF1 (see Fig. 9). In our example, most new operations are normalized or are very easy to normalize (see [37] for details). A more interesting example is

[2] robostar.cs.york.ac.uk/

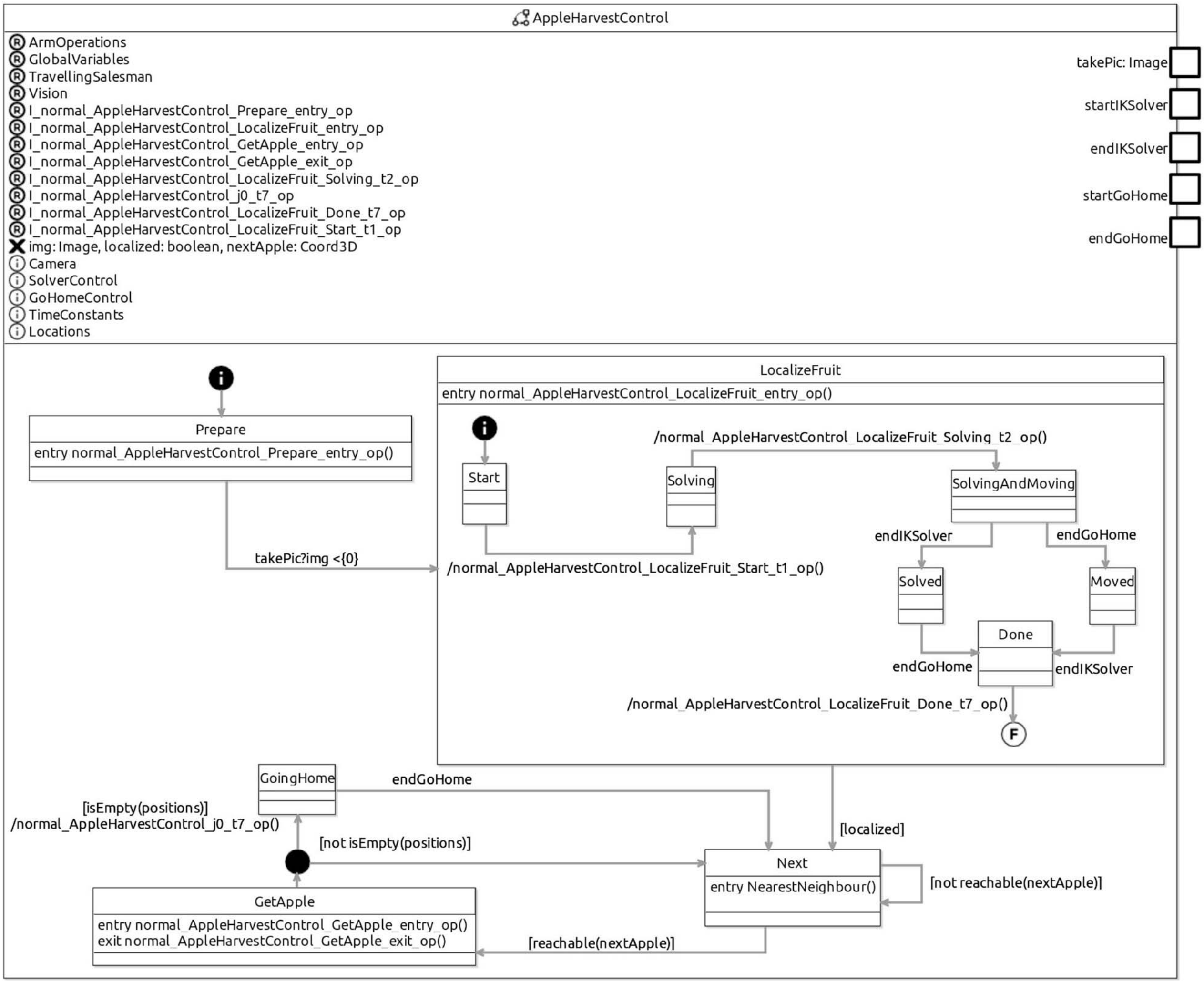


Figure 24. AppleHarvestControl after Step 2 of normSMB().

provided by the operation that we reproduce in Fig. 26; it is called normal_AppleHarvestControl_GetApple_entry_op(). To normalize it, we apply the procedure normAO() (Fig. 17).

Steps 1–4 of normAO() declare operations (and associated interfaces) that are used to capture the meaning of time primitives in the model and are already normalized. More interesting is Step 5, which applies the laws that remove the control flow from the action exhaustively. At first, only Law 11 applies, since the Statement is a deadline. The operations resulting from the application of this law are in Fig. 27. The interfaces used there are in [37].

The transformed definition in Fig. 27 for the operation normal_AppleHarvestControl_GetApple_entry_op() is now normalized. The new operation at the bottom of Fig. 27 has, however, the original Statement restricted by the deadline in a transition action in a composite state, and needs to be normalized. Since it is not an action operation, we first use normSMB(), which replaces the transition action with a call to the operation in Fig. 28. The relevant interface is in [37].

The action in the new operation in Fig. 28 is a sequence, and we apply Law 8 to decompose it. Fig. 29 shows the result of one application of Law 8 at the top, and, at the bottom, the result of its later exhaustive application after one application of Law 7 to remove the outer parentheses. (The spurious parentheses are introduced in the application of Law 11 to ensure that in s; g = true, the assignment to g follows s as stated. Without the parentheses, there is the possibility that g = true becomes captured, for instance, in the else branch of a conditional.)

At this point, there are three transitions with a wait statement. They are all nondeterministic, and we eliminate them with three applications of Law 13. The result is in Fig. 30. The order in which the laws are applied in Step 5 of normAO() does not matter. We can, for example, apply Law 13 as soon as an applicable transition is introduced by an application of Law 8.

This completes Step 5 of normAO(). The result, however, as shown in Fig. 30, is no longer an action operation. The decomposition of the original action introduced several transitions, and so normalization according to OPDNF1 does not hold. We then pursue normalization according to OPDNF2, which amounts to NCNF1 and NCNF2. Presence of sinceEntry(S) expressions and deadlines is not an issue, since they are removed by normSMB(), so NCNF2 is guaranteed. We may, however, need to introduce further operation Calls. In our example, most of the actions in the transitions in Fig. 30 are already Calls. In Step 6 of normAO(), we need to introduce just one more operation for the action in the transition to the final state.

The result is shown in Fig. 31, which is a normalized version of the operation in Fig. 28. This is obtained with an application of

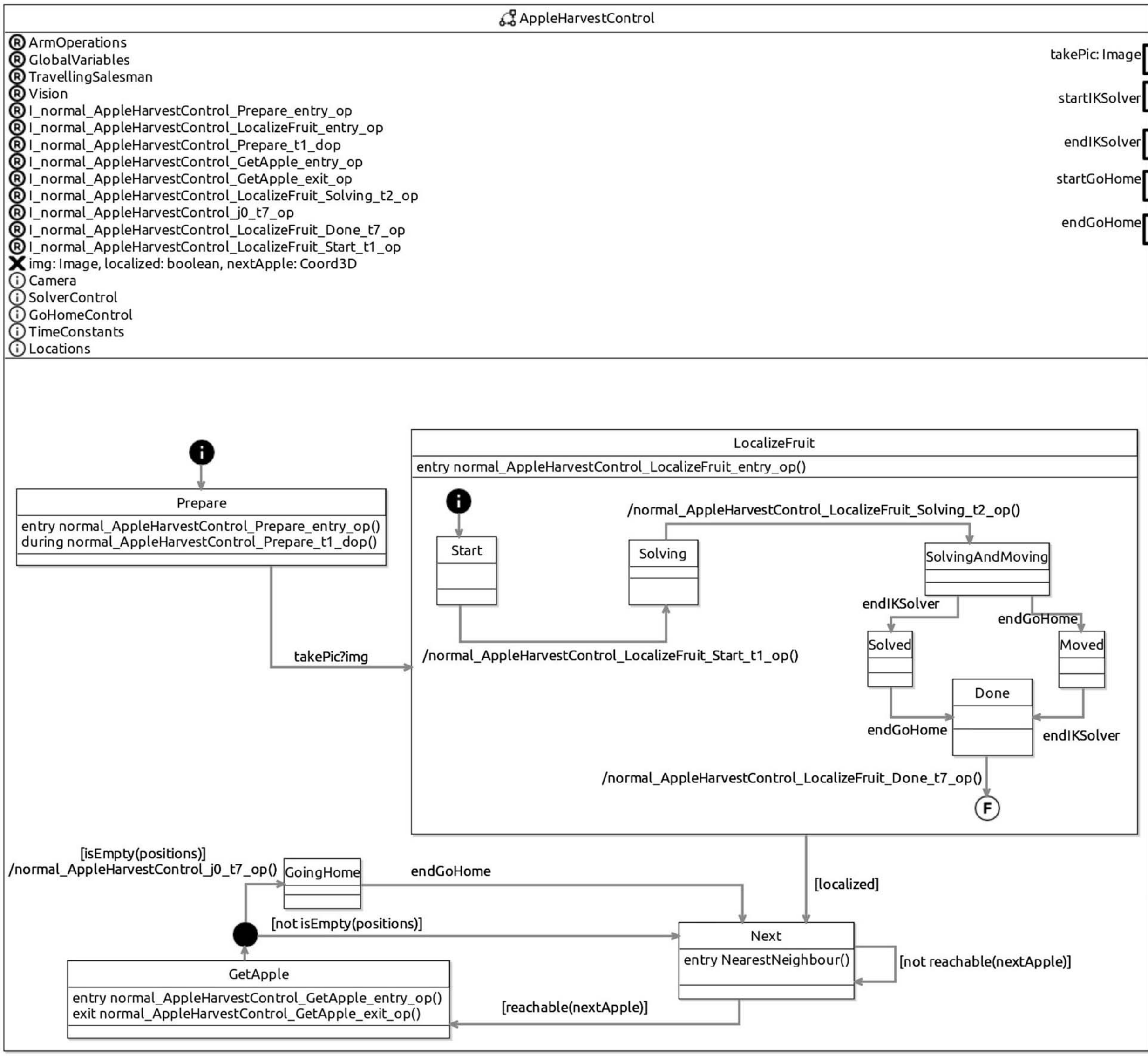


Figure 25. AppleHarvestControl after Step 5 of normSMB().

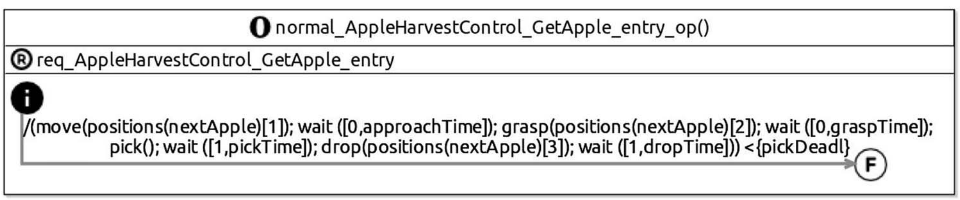


Figure 26. Example for use of normAO().

Law 3. The extra action operation called in the transition to the final state is in [37] ; it is normalized. In [37] we also present the new interface that is needed in this step.

The state machines presented here have been automatically generated from the original model (Fig. 2) using the tool described in Section 5.2. Our examples here illustrate how a model containing just normalized StateMachineDefinitions and OperationDefinitions can be obtained using our procedures.

5.1.2. *Normalization: open machines*

We now consider the normalization of an OpenStateMachine using the procedure normOM in Fig. 18. As said, for a BasicOpenStateMachine, first of all, in Step 1a, we apply a procedure normNC() that is very similar to normSMB() already illustrated. For example, if our starting point is an OpenStateMachine that is similar to the RoboChart machine AppleHarvestControl in Fig. 2, the result of Step 1a is a machine similar to that in Fig. 25. The difference is that an OpenStateMachine does not have a context. As we have discussed in Section 6, their semantics differ accordingly, as does the statement and proof of the relevant laws.

Step 1b of normOM applies decompBOM to both AppleHarvestControl itself and to the machine of its composite state LocalizeFruit. We illustrate first the application to AppleHarvestControl. The Step 1 is similar to that for the machine IKSolver presented in the

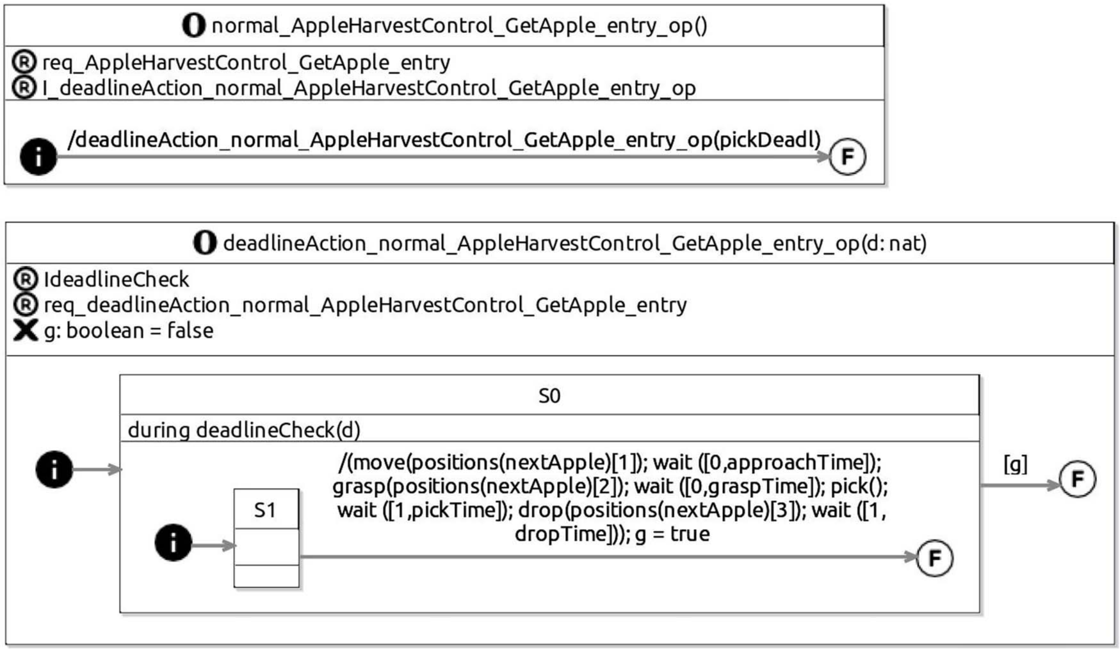


Figure 27. normAO(): application of Law 11.

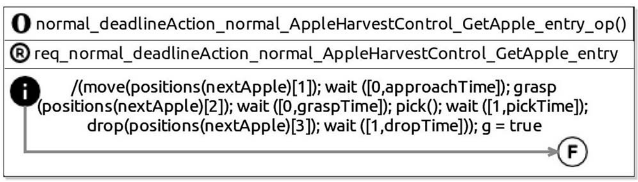


Figure 28. Action operation from normSMB().

previous section (see Fig. 21). AppleHarvestControl has, of course, a single EntryPoint: its initial junction, which, by a well-formedness condition, has a single transition that is not a self-transition. The target of that transition is the state Prepare, which is not a NodeNameRef. The application of Law 18 in Step 1a isolates the initial junction with a transition to a NodeNameRef connecting to Prepare. In the second machine in the composition, Prepare is an EntryPoint.

In Step 1b of decompBOM(AppleHarvestControl), a recursion leads to the application of decompBOM to the new machine with Prepare as a single EntryPoint. It has a transition to the composite state LocalizeFruit, which is not a NodeNameRef. Applying Law 18 to this machine, we get the result shown in Fig. 32.

Another iteration of the recursion splits LocalizeFruit and introduces a machine where the state Next is the single EntryPoint. Yet another recursion splits out Next and introduces a machine where GetApple is the EntryPoint. The new machines with Next and GetApple as EntryPoints are in Fig. 33. The self-transition of Next stays in the machine where this state occurs. The transitions to Next in the second machine become transitions to NodeNameRefs referencing Next.

In the next iteration, GetApple is isolated, with its transition to the Junction going to a new NodeNameRef, and the Junction becoming an EntryPoint of a new machine. The result is shown in Fig. 34. As mentioned before, Junctions have names, just like states. In diagrams, we normally hide such names. (Our tool, presented in the next section, generates fresh names for them automatically.) When, however, a junction is used as an EntryPoint, its name needs to be used, and it is shown in our example. In Fig. 34, the Junction is called j0 and its status as an EntryPoint in the new machine is marked by an i inside the Junction symbol.

The next iteration of the recursion applies to the new machine with j0 as EntryPoint, and isolates j0: see Fig. 35. Yet another machine is defined; it has the state GoingHome as EntryPoint. Since the only transition from GoingHome is to a NodeNameRef (to Next), and GoingHome is the single EntryPoint of this new machine, a recursive application of decompBOM to it has no effect. The procedure application decompBOM(AppleHarvestControl) is now finished.

For LocalizeFruit, after four iterations of the recursion, we get the machines in Fig. 36. The rightmost machine (at the bottom of Fig. 36) has two EntryPoints: the states Solved and Moved. The fifth iteration splits that machine into three as shown in Fig. 37: one for each EntryPoint and one with the rest of the nodes. Since Solved and Moved have each a single transition, both to the state Done, the extra machine has again a single EntryPoint: Done.

The next iteration of the recursion separates out Done, leaving just the Final state. Figure 38 shows the result. The diagram for the last machine reveals the name of the Final state, and uses an i to indicate that it is an EntryPoint. A recursive application of decompBOM to this machine has no effect. The normalization of LocalizeFruit is concluded.

Next we describe our implementation in RoboTool of the normalization procedures.

5.2. RoboTool

For a preliminary validation of our laws and procedures, we have implemented them. We have used the model-transformation engine Epsilon[3] [38], an open-source framework with a set of languages and facilities for the management and development of models with rich Eclipse integration. We have integrated our implementation with RoboTool[4] , a modelling and verification tool for RoboChart.

3 www.eclipse.org/epsilon/
4 robostar.cs.york.ac.uk/robotool/

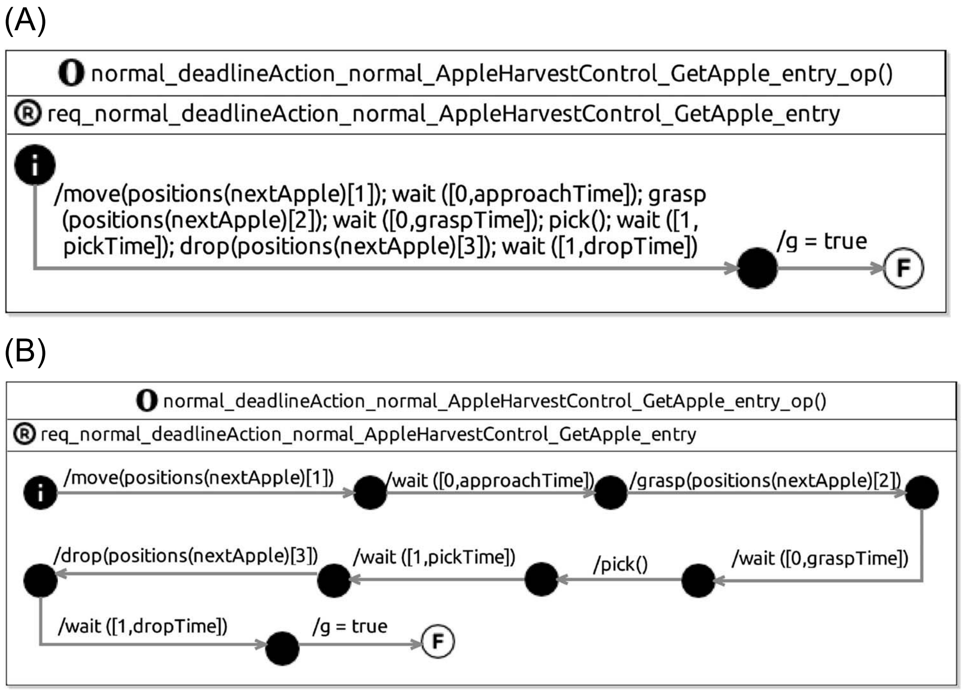


Figure 29. Law 8: (A) once and (B) exhaustively.

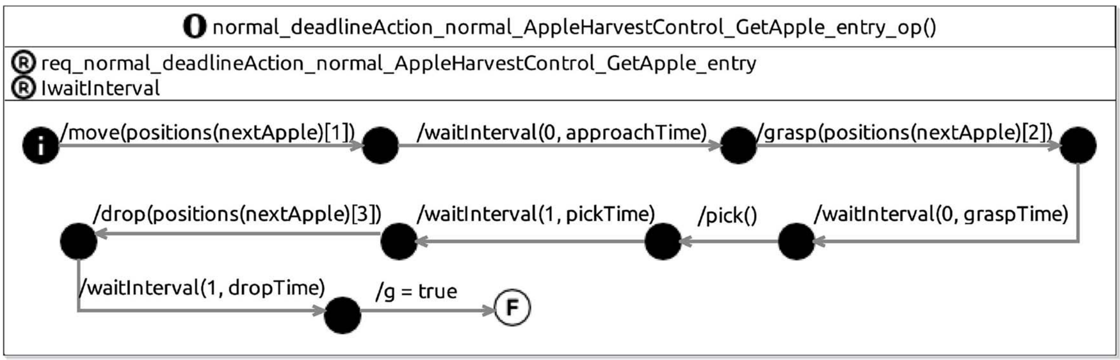


Figure 30. Law 13, three times.

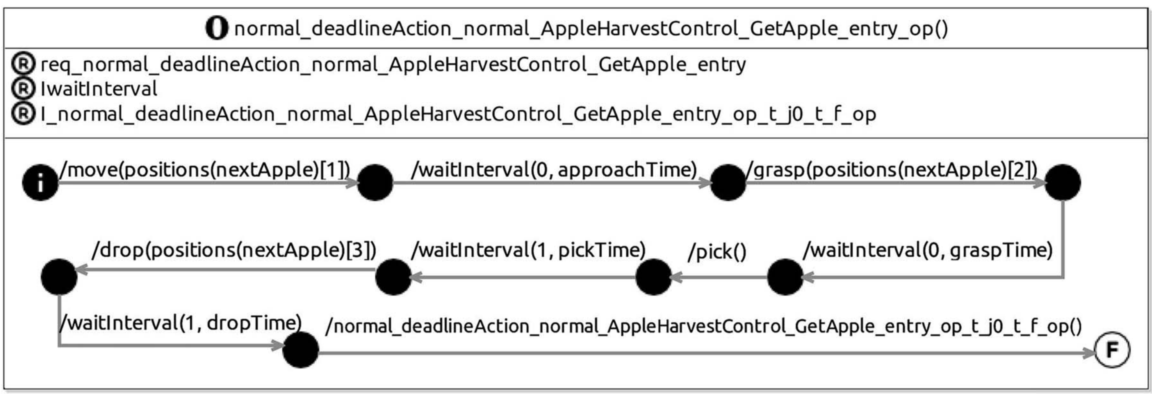


Figure 31. normAO(): final result.

RoboTool is a set of Eclipse plug-ins that provides textual and graphical editors, implemented using the Xtext[5] and Sirius[6] frameworks. RoboTool generates, automatically, a (tock-)CSP and a reactive modules semantics [39] for RoboChart models. It is integrated with the FDR [40] and PRISM [41] model checkers to prove behavioural and quantitative properties.

With our implementation, and the testing it has enabled, we have established that the laws are well-typed, and have gathered evidence for the correctness of the procedures. Our test suite[7]

5 eclipse.org/Xtext/
6 www.eclipse.org/sirius/
7 github.com/UoY-RoboStar/robochart-normalization-tests

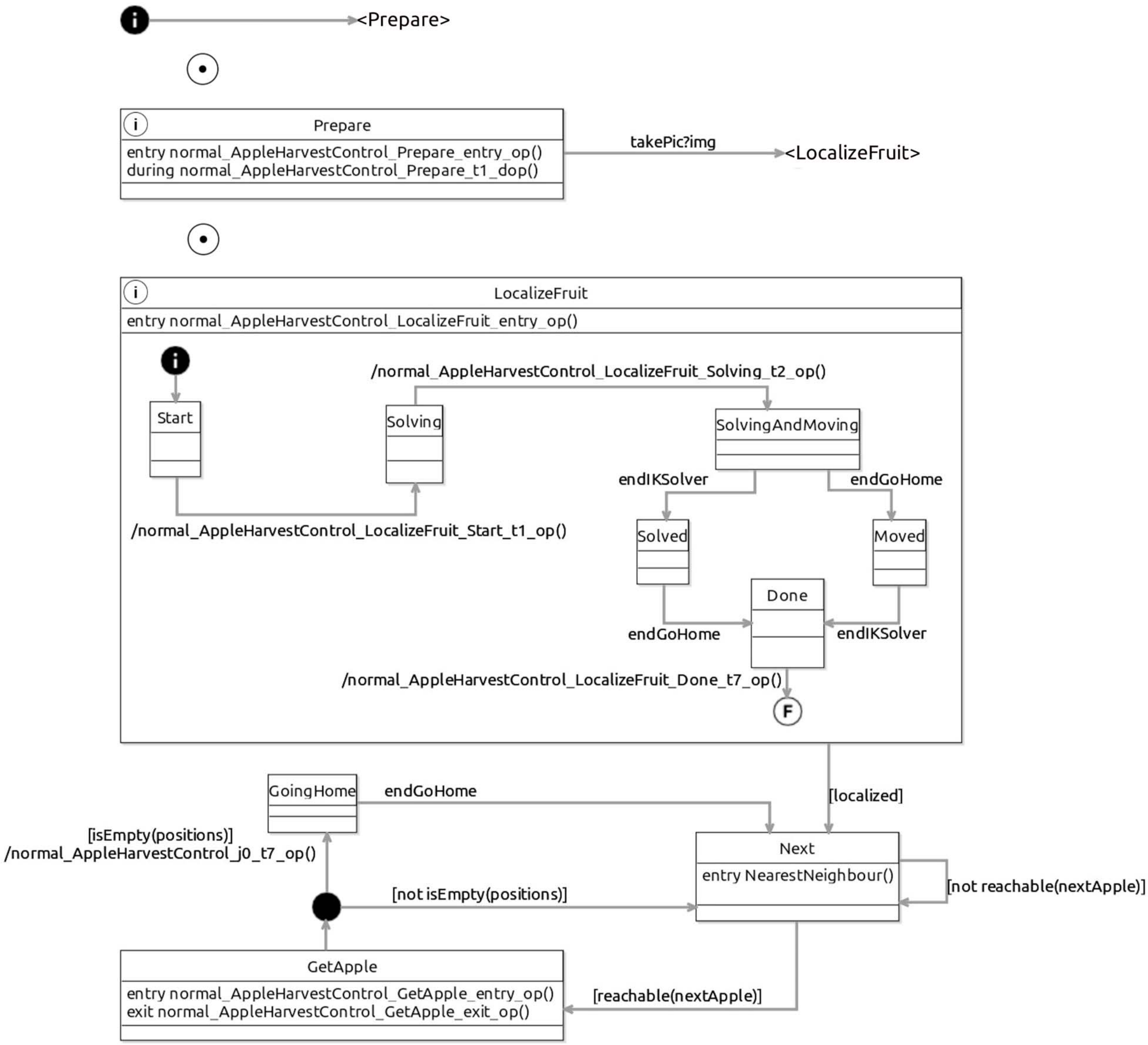


Figure 32. decompBOM(AppleHarvestControl): after 2nd recursion.

contains models that, together, include at least one instance of every class of the metamodels and whose normalization requires the application of every law presented here. A test execution includes the comparison of the tock-CSP semantics of the original and normalized machines.

In fact, the laws and procedures described in the previous section are the result of our experience with the development and use of our tool. This effort has led to the identification of missing declarations in the body and provisos of some laws (such as Laws 2 and 6) that ensure that the machines that are defined by their application are well formed. Our experiments have also revealed early the need for adjustments in the order in which the laws are applied. For example, for the procedure normAO(), the experiments have made apparent the need for the extra Step 6.

Each law is implemented as a transformation pattern using the Epsilon Pattern Language (EPL)[8] , a language based on pattern matching for specifying and detecting structural patterns for model-to-model transformation. The implementation of each law captures a structural pattern of interest (based on the left-hand side of the law as presented here) to produce in-place modifications of the model (reflecting the result of the law application defined by its right-hand side). As an example, we present in Fig. 39 a fragment of the EPL pattern that implements Law 2.

An EPL pattern is defined by named and typed elements of the metamodel, called roles. For Law 2 (see Fig. 39), we define three roles (lines 2-4): a machine body (`smb`), a state (`S`) of such a machine, and, finally, an action (`s`) of `S`. The types prefixed by RoboChart! refer to classes of the RoboChart metamodel. Types without such a prefix are native to Epsilon.

The implementation of Law 2 creates the new operation, `OpS`, and sets its name (lines 9–10). As defined by Law 2, `OpS` contains a single Transition from an Initial junction to a Final state. The junction and the state are recorded in the variables `OpS_I` and `OpS_F` defined on lines 11–14. These variables are used to set the source and target of the transition `OpS_Tr` (lines 18–19). The action of this transition is the original action of the state `S` (line 20). The new elements (that is, `OpS_I`, `OpS_F` and `OpS_Tr`) are, finally, added to the set of `nodes` and `transitions` of `OpS` (lines 22–23).

The signature of `OpS` is defined on lines 27–28 for inclusion in the interface `IOpS` defined on lines 29–31. On line 32, we

8 www.eclipse.org/epsilon/doc/epl

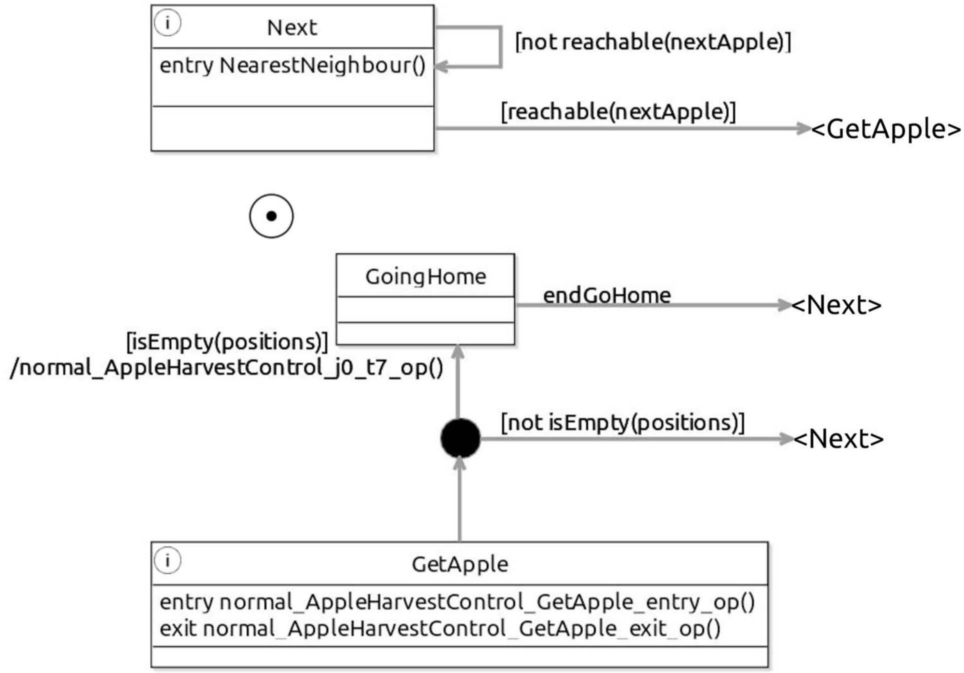


Figure 33. decompBOM(AppleHarvestControl) After fourth recursion.

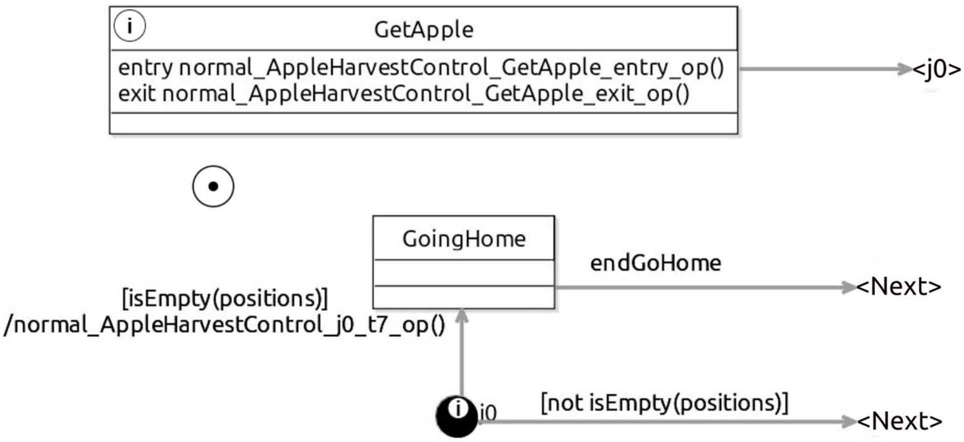


Figure 34. decompBOM(AppleHarvestControl) After fifth recursion.

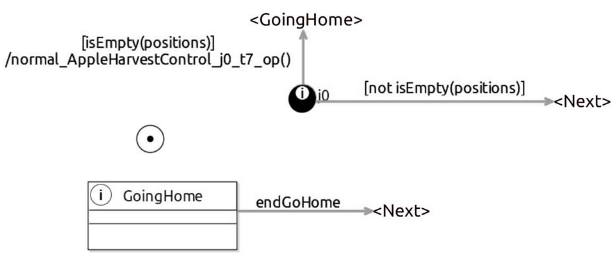


Figure 35. decompBOM(AppleHarvestControl) After sixth recursion.

add `IOps` to the set `rInterfaces` of required interfaces of the StateMachineBody. The other interfaces `req_s` and `defEvents_s` in Law 2 are defined and included in a similar way (omitted in Fig. 39).

The main goal of Law 2 is to use an operation Call to replace an action of a state. This is effectively implemented in lines 36–38, where a Call (`OpS_Call`) is created for the new operation, and the original action of the state `s` is defined to be this Call.

We have also implemented the normalization procedures for a StateMachineBody, normSMB() and for action OperationDefinitions, normAO(). For OpenStateMachines, decompBOM(openM), the core of normOM(openM) is implemented. The additional procedure normNC() used in normOM(openM) is not implemented because it is similar to normSMB().

Each procedure is implemented as an ANT-based Epsilon workflow[9] , a mechanism called target for performing model management and transformation activities. In our implementation, each target corresponds to a normalization procedure. The execution of a workflow performs a sequence of EPL tasks, specified using an epsilon.epl clause. Each task corresponds to a step of the target procedure as defined in Section 4. As illustration, we present in Fig. 40 a sketch of the implementation of normSMB(). (A workflow is defined using an XML-based notation.)

[9] ant.apache.org

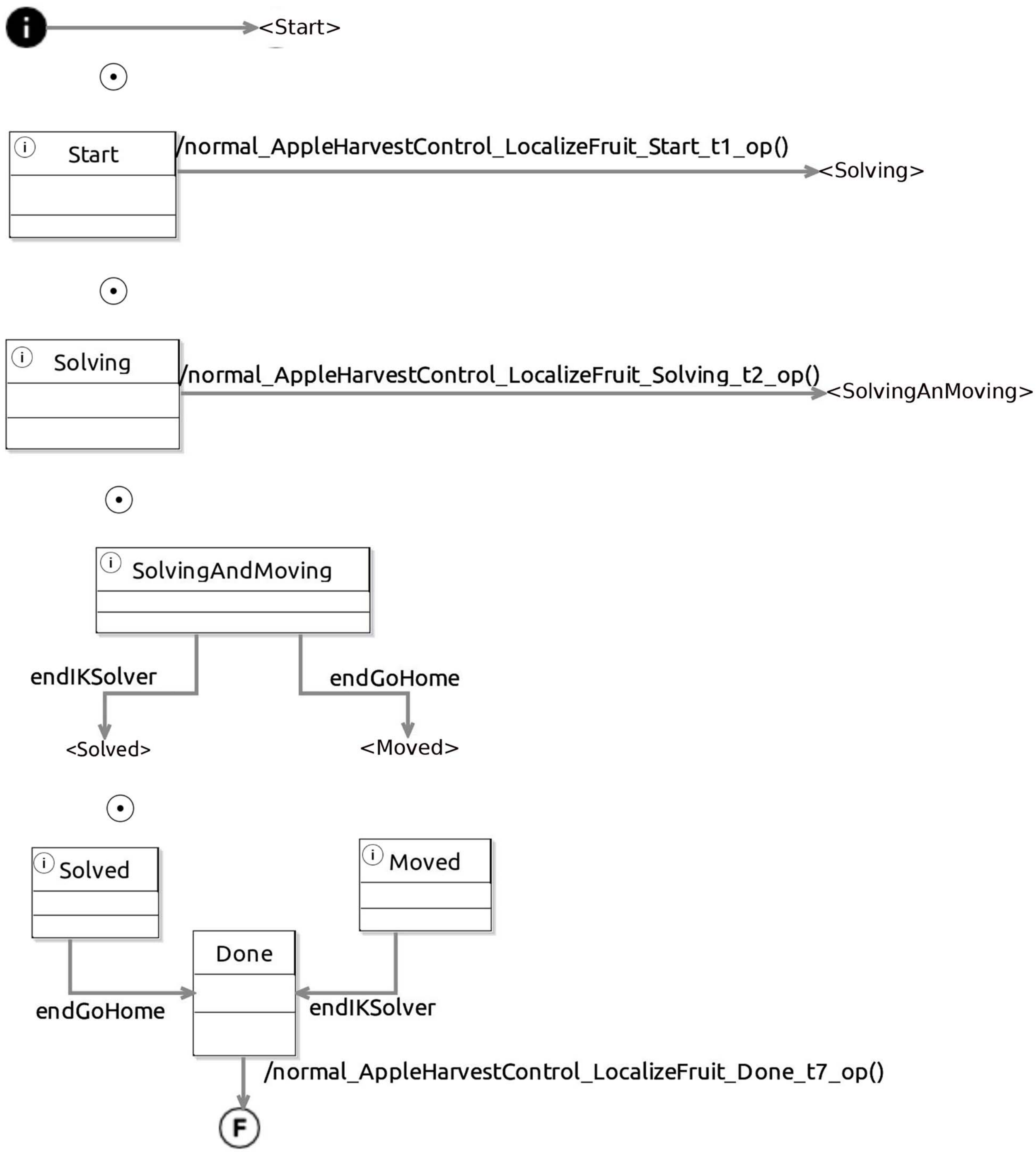


Figure 36. decompBOM(LocalizeFruit): after fourth recursion.

As depicted in Fig. 40, the first task in the workflow for normSMB() implements Step 1, as specified by the `uri` attribute in its `epsilon.epl` clause. Internally, `Step1.epl` calls Law1 1 (elimsinceEntry). `Step1.epl` is repeated as long as successful matches of the pattern in that law in the model (`RoboChart`) are found, since the clause `repeatWhileMatches` is set to `true`. When there are no more matches, the task is finished, and the subsequent task is executed. Step 2 in the definition of normSMB() (see Fig. 16), however, is applied while there are actions that are not a Call. In ANT, there is no direct mechanism to implement tactics for iterative applications of transformations based on patterns other than those in the laws. We have, therefore, encoded the termination condition for each law where relevant. So, in addition to the code fragment presented in Fig. 39, our implementation of Law 2 actually includes the following match condition:

```
match : nots.action.isTypeOf(RoboChart!Call)
```

This ensures that Law 2 is applied only when the action is not a Call. The target `normSMB` ends after the task that calls Step 5 and its execution is completed.

As an optimization in our tool, given a model, we apply in parallel the procedures described above to all machines and

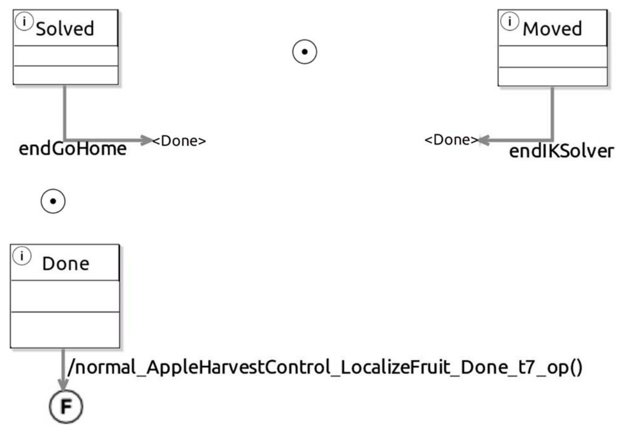


Figure 37. decompBOM(LocalizeFruit): after fifth recursion.

Figure 38. decompBOM(LocalizeFruit): after sixth recursion.

operations. This includes the operations that might be introduced as the result of an application of a law. In the case of Law 11, which is used in Step 5 of normAO(), it introduces an operation deadlineAction˙s(d:nat) to which we need to apply normSMB(). As part of our optimization, the definition for deadlineAction˙s(d:nat) that we introduce is already the result that would be obtained by normSMB().

In RoboTool, the implemented procedures are encapsulated by a plug-in, called *NormalForm Generator*, and are applied in the following order: normSMB(), normAO(), and decompBOM(). This also reflects the optimization previously described, which applies the procedures to machines and operations. So, when a RoboChart model is loaded, the *NormalForm Generator* can be invoked by clicking on the menu item `NormalForm` in its toolbar, as shown in Fig. 41. The normalized RoboChart and OpenMachine models are stored in the project directories `/normalForm/robochart` and `normalForm/openMachine`, respectively.

The examples presented here, and many others, have been developed using RoboTool.

5.3. Applications

A model or program normalization procedure is an effective technique to establish a notion of (relative) completeness of a set of algebraic laws, as we have already pointed out and illustrated in Section 4.

Moreover, additional applications of such laws arise from the benefit of using normalization as a preliminary step in the translation into other notations. Some other transformation techniques, however, like refactoring, involve law applications with a different purpose: improving readability and reuse, and reducing complexity of models. In this section, we briefly illustrate law-based refactoring transformations.

```
pattern intro_call_for_act_state
 smb : RoboChart!StateMachineBody,
 S : RoboChart!State from: nodes(smb),
 s : RoboChart!Action from: S.actions {
do {

 ...

 var OpS = new RoboChart!OperationDef();
 OpS.name = setName(smb,S,s);
 var OpS_I = new RoboChart!Initial();
 OpS_I.name = "i";
 var OpS_F = new RoboChart!Final();
 OpS_F.name = "f";

 var OpS_Tr = new RoboChart!Transition();
 OpS_Tr.name = "t";
 OpS_Tr.source = OpS_I;
 OpS_Tr.target = OpS_F;
 OpS_Tr.action = s.action;

 OpS.nodes.addAll(Set{OpS_I,OpS_F});
 OpS.transitions.add(OpS_Tr);

 ...

 var OpSig = new RoboChart!OperationSig();
 OpSig.name = OpS.name;
 var IOpS = new RoboChart!Interface();
 IOpS.name = "I_" + OpS.name;
 IOpS.operations.add(OpSig);
 smb.rInterfaces.add(IOpS);

 ...

 var OpS_Call = new RoboChart!Call();
 OpS_Call.`operation` = OpSig;
 s.action = OpS_Call;
  }
}
```

Figure 39. Implementation of Law 2.

```
<target name="normSMB">
    <epsilon.epl uri="Step1.epl"
        repeatwhilematches="true">
        <model ref="RoboChart"/>
        ...
    </epsilon.epl>

    ...

    <epsilon.epl uri="Step5.epl"
        repeatwhilematches="true">
        <model ref="RoboChart"/>
        ...
    </epsilon.epl>
</target>
```

Figure 40. Implementation of normSMB().

To avoid bias, we use an existing RoboChart model with some slight simplifications: a robot in a swarm acting under the Alpha

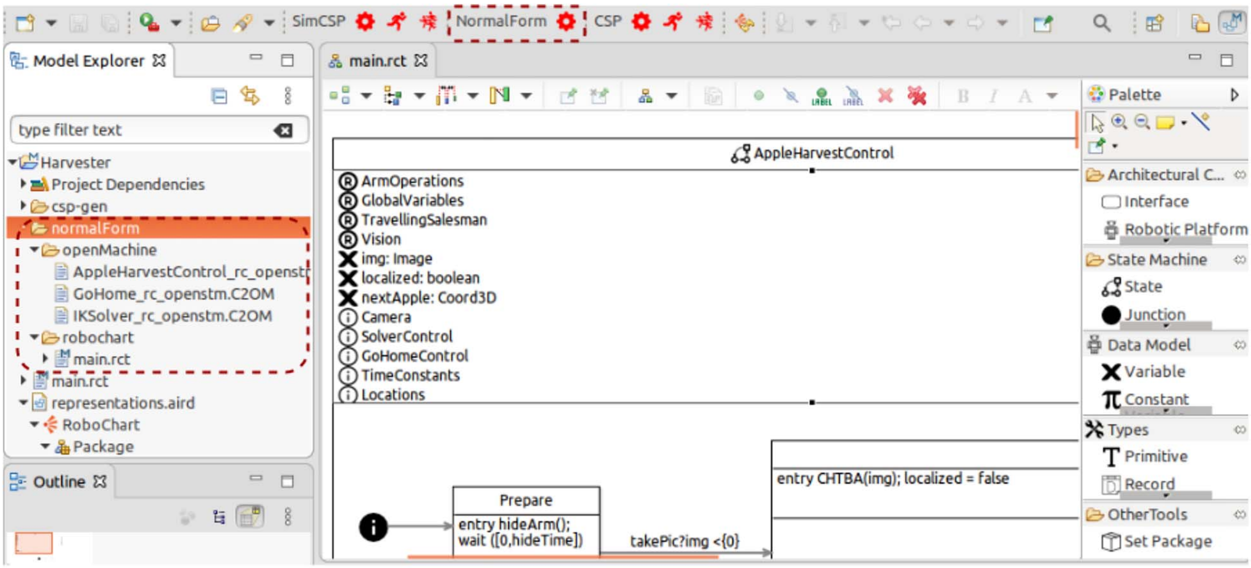


Figure 41. RoboTool interface.

Algorithm [42]. This example includes one machine capturing the behaviour of the robot's movement, and another modelling the communication with the other robots. Our focus is on the Movement machine: see Fig. 42.

Its initial transition includes an action that resets the clock MBC, and targets the composite state MovementAndAvoidance. Entering this state happens only after executing the entry action (move(lv,0)) of the state Move. This operation is parametrised by a linear and an angular velocity, in this order.

When an occurrence of an obstacle is detected, if less than MB-360/av time units have passed, the transition from Move to Avoid is triggered. In this case, the entry action of Avoid (a conditional that determines an angular motion, based on the robot's position) is executed, and a period of time must pass (as stated in the wait action) before returning back to Move.

Another control path is via the transition from MovementAndAvoidance to Turning; it is enabled when at least MB time units have passed since the last clock reset. The action associated with this transition is a sequence formed of the reset of the MBC clock, an input action that reads the number n of neighbouring robots, and an assignment of the value false to a control flow flag turned. Once in the state Turning, the robot is required to either turn 180 degrees (top transition to final state) if n is below a threshold alpha or, otherwise, perform a random turn (bottom transition). In any case, the control flag takes the value true to allow the triggering of the transition back to MovementAndAvoidance.

There are opportunities to both modularize and simplify this model. For example, there are four occurrences of a call to move followed by a wait statement: two in the entry action of Avoid and two on the transitions of Turning; the differences are only in the values of the arguments. Therefore, there is benefit in introducing a new operation, called say turn, that encapsulates this pattern of use of move and wait, and calling this operation with the appropriate arguments, instead of duplicating actions across the model.

To replace the two occurrences in the entry action of Avoid, we first note that the then and else branches of the conditional include only the invocation of the operation move. A single wait statement comes after the conditional. This is a nice example that illustrates the need for combining complementary sets of algebraic laws in practical applications of model and program transformation. In this particular context, we need a simple law of the conditional statement that allows distributing a statement after a conditional into its branches (at the end of the then and else branches):

if c then s1 else s2 end; s3

=

if c then s1;s3 else s2;s3 end

Applying this law, we can rewrite the conditional in the entry action of Avoid as in Fig. 43. A seminal paper includes several such laws of programming [7]; they are not our focus here, but are complementary to our laws of state machines. The combined use of these sets of laws justifies very expressive transformation strategies.

After applying the above law, we can apply Law 2 to replace the sequential actions in the then and else branches of the conditional in Fig. 43 with calls to turn. The result is in Fig. 44. The first call can be introduced with a direct application of Law 2, introducing the interface, the definition of turn, and the call. The second call can be introduced using a slightly simplified version of the law that introduces only the call to the already declared turn operation.

Similarly, to refactor the actions in Turning, we first isolate the call to move followed by the wait statement into an action of a separate transition. This is justified by two applications of Law 8. Afterwards, we apply Law 3 to replace these sequential actions with calls to turn, in the same way as performed in the entry action of Avoid. The result is in Fig. 45.

Now we combine the actions in the top and bottom transitions of Turning back into sequential actions. This is justified by two applications of Law 8, from right to left. The result is in Fig. 46.

At this stage, we can combine the two transition actions of Turning into a conditional action. This is justified by applying Law 9, also from right to left. Furthermore, the transition from the initial junction to a junction can be eliminated by an application of Law 10, once more, from right to left. The effect of these two transformations is presented in Fig. 47.

The resulting model of the Movement machine after all these transformations is depicted in Fig. 48. It is possible to carry out

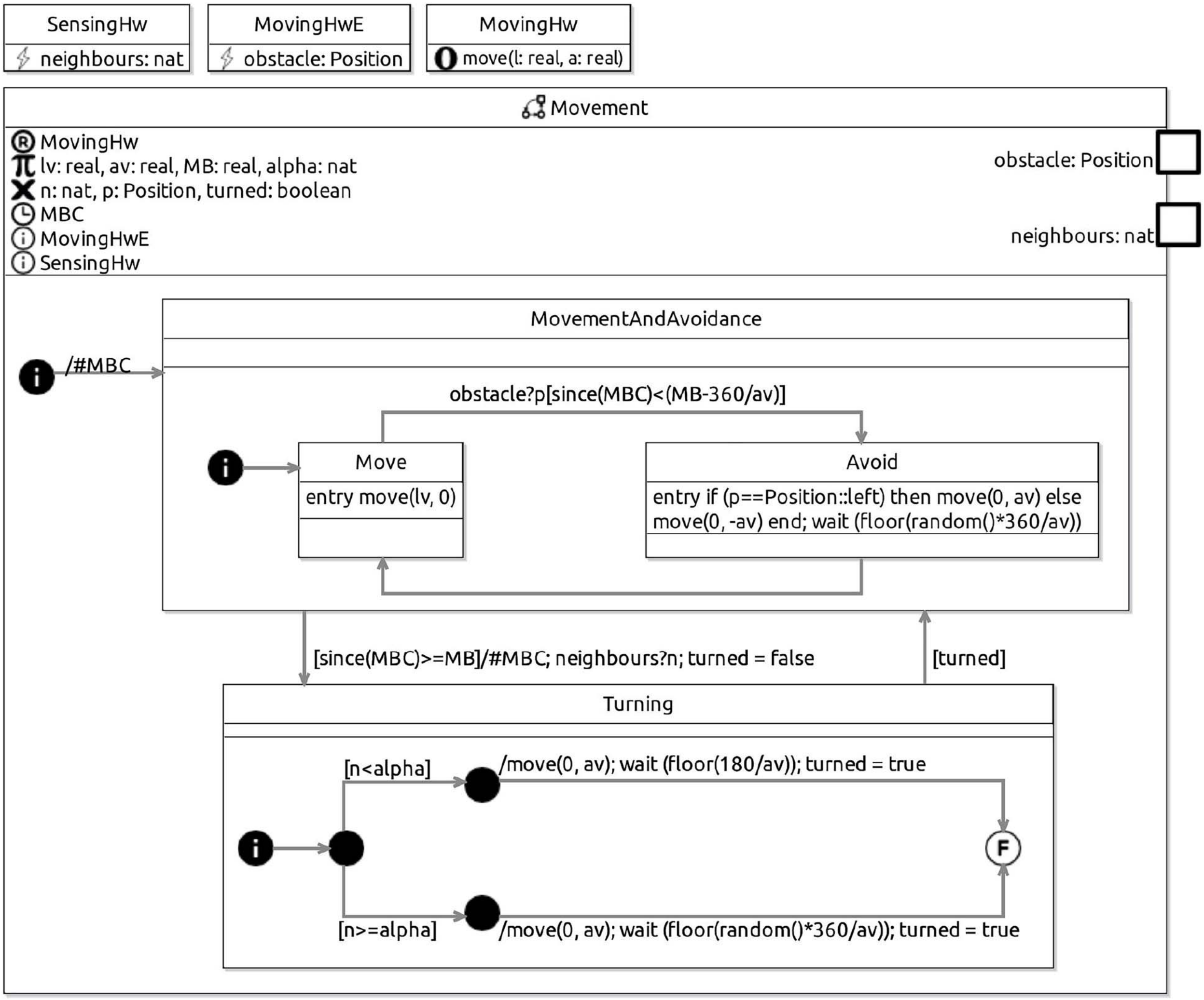


Figure 42. Alpha Algorithm.

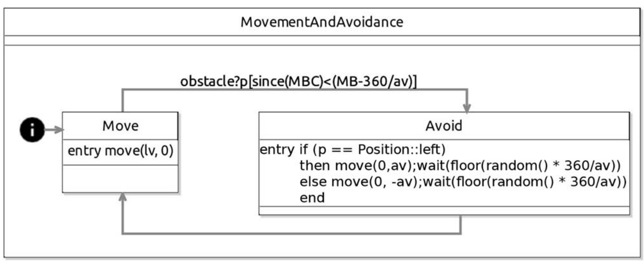


Figure 43. Alpha Algorithm - Step 1.

the transformations that are based on the laws presented in this paper using our encoding of these laws in RoboTool and Epsilon facilities to select and direct the application of laws[10] . Encoding of the laws of programming is not in the scope of our work, but the facilities of Epsilon make this a simple task.

In our ongoing work, we are using normalization to facilitate transformation of RoboChart models into simulation models (described using another diagrammatic notation called RoboSim [43]). Any such technique for RoboChart and open machines is simplified by adding normalization as a first step, and then proceed with transformations that need to consider just normalized models. These normalized models exhibit only a very reduced number of patterns for transformation, characterized by the Definitions 3.1, 3.2, and 3.3 of the normal forms.

For all the above applications, soundness of the laws is paramount. We address this point next.

6. SOUNDNESS

Soundness of the normalization is justified using the formal semantics presented next in Section 6.1. Proofs of soundness of selected laws are sketched in Section 6.2. Complete proofs are available in [37].

6.1. Semantics

In this section, we define a discrete-time semantics in *tock*-CSP [44] for RoboChart and open state machines. Besides providing a novel semantics for open machines, we give a new compositional account of the meaning of operations in RoboChart. This is important for establishing soundness of our laws, which make extensive use of operations to encapsulate behaviour.

The process algebra *tock*-CSP is a timed variant of CSP [35]; it is part of a large family of notations for specifying concurrent

10 At github.com/UoY-RoboStar/robochart-normalization.

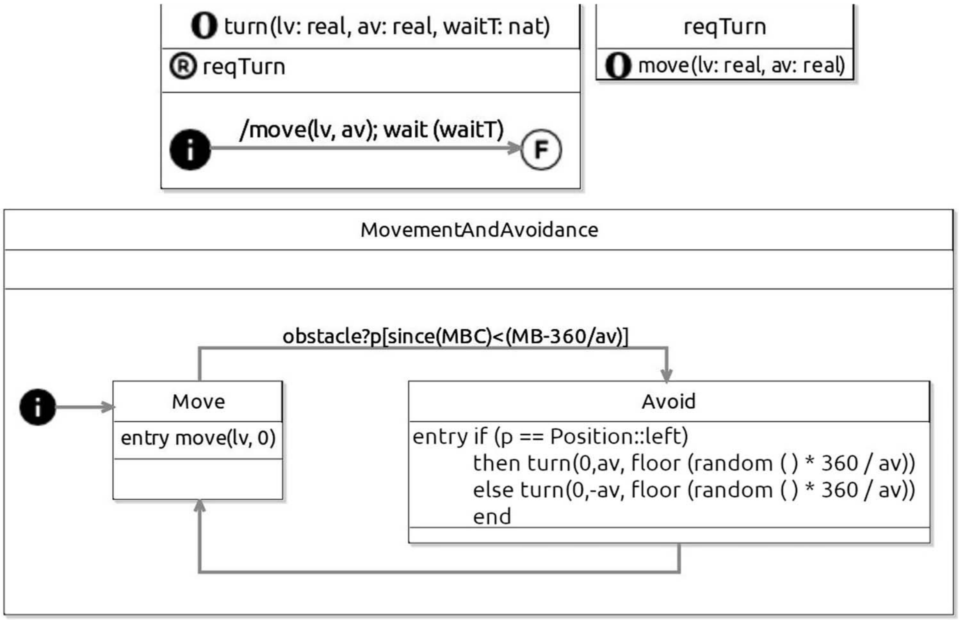


Figure 44. Alpha Algorithm - Step 2.

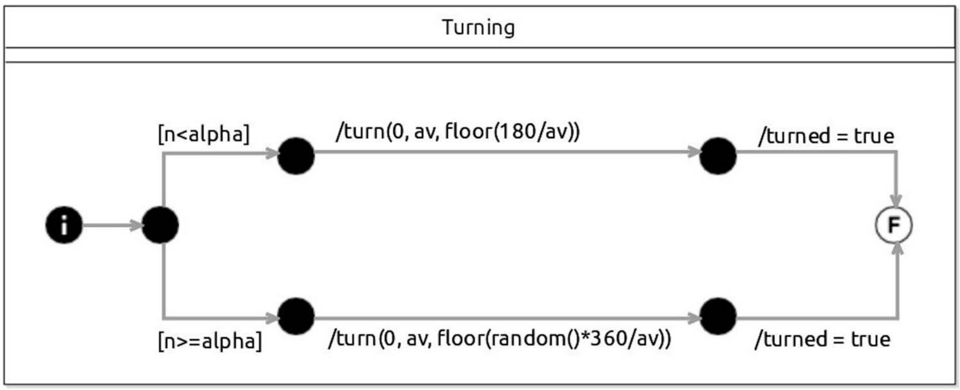


Figure 45. Alpha Algorithm - Step 3: State Turning.

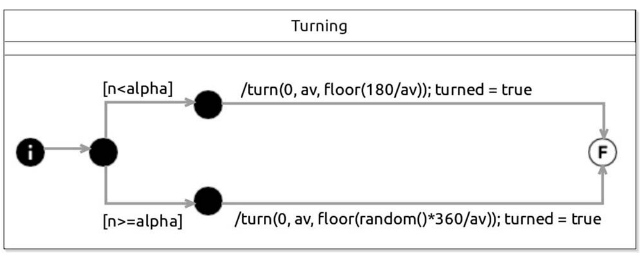


Figure 46. Alpha Algorithm - Step 4.

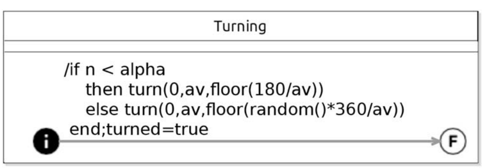


Figure 47. Alpha Algorithm - Step 5.

systems, including CCS [45], Pi-Calculus [46] and ACP [47]. CSP is distinctive in its denotational semantics, giving rise to notions of refinement useful for stepwise development. Processes specify patterns of interaction via synchronization on channels, taking into account aspects such as (non)determinism, deadlock, and termination. Process definitions can also be made via parallel composition. Communications between parallel processes or with the environment are instantaneous, atomic events, that can carry values: inputs and outputs. The dialect *tock*-CSP, in addition, allows processes to specify time budgets and deadlines using a special event called *tock*.

In Table 1 we summarize the *tock*-CSP operators that we use in our work. To illustrate the notation we present a simple example of a one-place timed buffer.

Example 1.

$$TB = in?x \rightarrow (TB \,\square\, (Wait(1); out!x \rightarrow TB))$$

The buffer is defined by the process *TB*. Initially it is prepared to receive (?) a value *x* on the channel *in* via a prefixing ($in?x \rightarrow$), and then offers an external choice ($\square$) to the environment between accepting a new value, via the recursion on *TB*, or delaying the output (!) of the current value. The delay of one time unit (*Wait*(1)) is sequentially composed (;) with a prefixing on *out*. An external choice is resolved by the environment synchronizing on events,

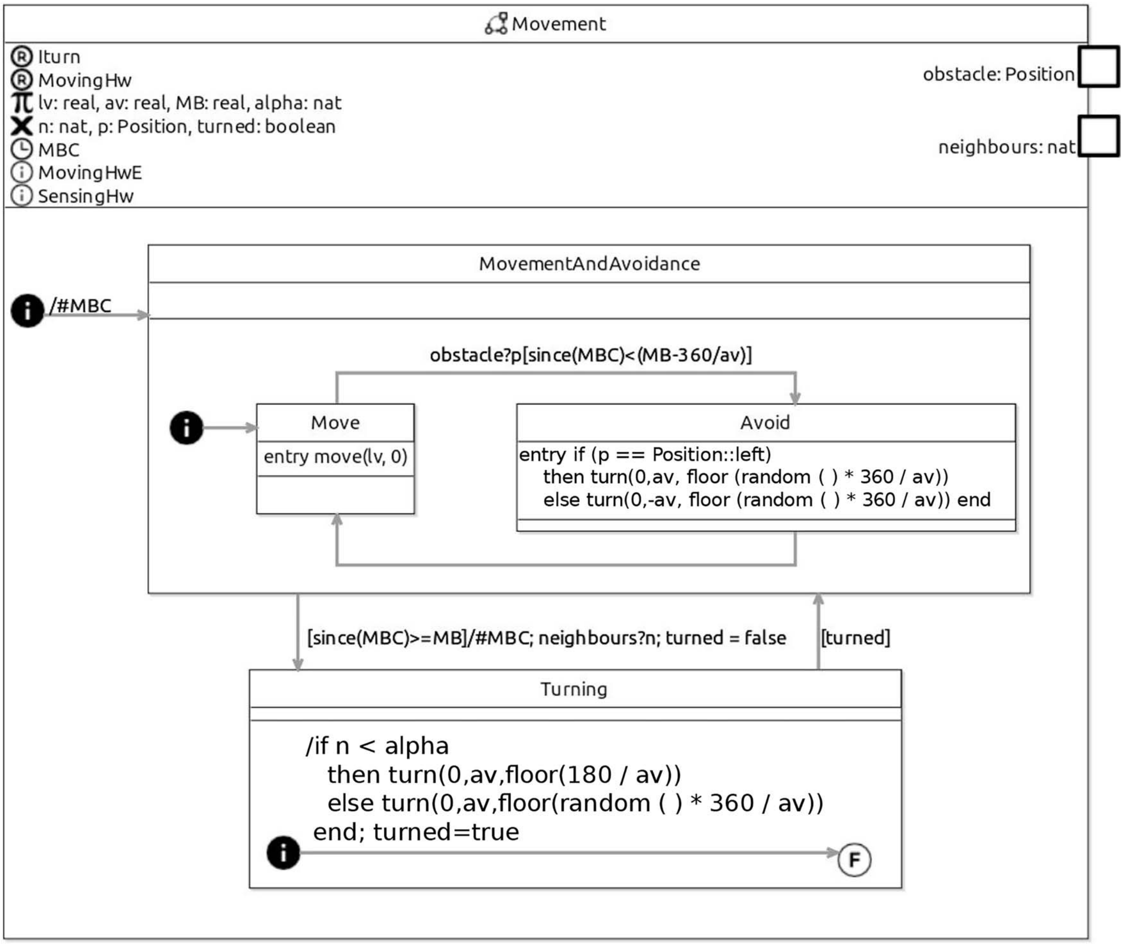


Figure 48. Alpha Algorithm - Refactored model.

or by termination. Here *in?x* is syntactic sugar for accepting events *in.x*, where *x* ranges over the type of the channel *in*, left unspecified in this example. The prefixing on *out*! *x* communicates the value of *x* as introduced into context by *in?x*.

The denotational semantics of *tock*-CSP [44] forms a complete lattice under the refinement order $\sqsubseteq$. Here $P \sqsubseteq Q$ means that Q refines P, and is true exactly when every timed behaviour of Q, denoted by a trace that records interactions, refusals, and passage of time, is also a behaviour of P. The semantics of a recursive process, such as *TB*, is the greatest fixed point of *TB*, taken as a function from processes to processes.

Section 6.1.1 gives an overview of our semantics. Section 6.1.2 details the semantics of RoboChart state machines, and Section 6.1.3 of open machines.

6.1.1. *Overview*

The structure of our *tock*-CSP semantics is illustrated in Fig. 49. It is formalized by semantic functions that define, for the various elements of the metamodel for RoboChart and open machines, *tock*-CSP processes or terms used to specify such processes. The semantic functions are used to define the notions of equality that we use in the laws, based on mutual refinement of the processes defined by the functions.

At the core of the semantics of a state machine is a parallel composition of processes that capture the semantics of its Nodes, taking into account any transitions with deadlines, and of processes for the Transitions. These processes synchronize on the CSP channels below that model the control flow. In what follows, we use grey and underlined text to indicate terms of the metanotation used to define the semantics.

- *end*, used to control termination of state machines, and operations, when a Final state is activated;
- *enter*.id(n), used to activate a node n, uniquely identified by a metafunction id;
- *entered*.id(n), used to indicate completion of the activation of a node n;
- *exit*, used to deactivate the currently active state;
- *exited*, used by the currently active state to indicate it has completed its deactivation;
- *interrupt*.id(n), used to interrupt the execution of n, for example, when a transition is triggered.

Although the semantics of Nodes is defined by a parallel composition, the control flow of a NodeContainer is sequential, and so only one of its Nodes is active at a time, as constrained by the parallel composition with the process semantics of its Transitions.

Activation of a State n leads to the execution of its entry action and the activation of its substates, if there are any. As said, completion is indicated via synchronization on the channel *entered*.id(n) used to:

- communicate to (the process for) a parent State p of n, that its child State n has completed activation;
- enforce any deadlines on outgoing transitions from n, as modelled by a Transition deadlines process as indicated in the innermost box in Fig. 49;

Table 1. *tock*-CSP operators, with basic processes at the top, followed by composite processes: P and Q are metavariables that stand for processes, d for a numeric expression, e for an event, a and c for channels, x for a variable, I for a set, v for an expression, g for a condition, and X for a set of events. For a channel c, $\{\!|c|\!\}$ is a set of events; if c is a typed channel then events are constructed using the dot notation, so that $\{\!|c|\!\} = \{\!|c.v_0, \ldots, c.v_n|\!\}$, where v_i ranges over the type.

Process	Description
$Skip$	**Termination**: terminates immediately
$Wait(d)$	**Delay**: terminates exactly after d units of time have elapsed
$Stop$	**Timed deadlock**: no events are offered, but time can pass
$Stop_U$	**Timelock**: no events are offered and time cannot pass
$e \rightarrow P$	**Prefix operator**: initially offers to engage in the event e while permitting any amount of time to pass, and then behaves as P
$a?x \rightarrow P$	**Input prefix**: same as above, but offers to engage on channel a with any value, and stores the chosen value in x
$a?x : I \rightarrow P$	**Restricted input prefix**: same as above, but restricts the value of x to those in the set I
$a!v \rightarrow P$	**Output prefix**: same as above, but initially offers to engage on channel a with a value v
if g **then** P **else** Q	**Conditional**: behaves as P if the predicate g is true, and otherwise as Q
$P \Box Q$	**External choice** of P or Q made by the environment
$P \sqcap Q$	**Internal choice** of P or Q made non-deterministically
$P; Q$	**Sequence**: behaves as P until it terminates successfully, and, then it behaves as Q
$P \setminus X$	**Hiding**: behaves like P but with all communications in the set X hidden
$P \mid\mid\mid Q$	**Interleaving**: P and Q run in parallel and do not interact with each other
$P \mid[X]\mid Q$	**Generalized parallel**: P and Q must synchronize on events that belong to the set X, with termination occurring only when both P and Q agree to terminate
$P[\![a \leftarrow c_1, \ldots, a \leftarrow c_i]\!]$	**Renaming**: replaces uses of channel a with channels c_1 to c_i in P
$P \triangle Q$	**Interrupt**: behaves as P until an event offered by Q occurs, and then behaves as Q
$P \triangle_d Q$	**Strict timed interrupt**: behaves as P, and, after exactly d time units behaves as Q
$P \; \Theta_X \; Q$	**Exception**: behaves as P until P performs an event in X, and, then behaves as Q
$[\mid X \mid]\, i : I \bullet P(i)$	**Replicated generalized parallel**: behaves as $P(i)$ in parallel for all i in I synchronizing in X
$\mid\mid\mid\, i : I \bullet P(i)$	**Replicated interleaving**: behaves as $P(i)$ interleaved for all i in I
$\Box\, i : I \bullet P(i)$	**Replicated external choice**: offers an external choice over processes $P(i)$ for all i in I
$\sqcap\, i : I \bullet P(i)$	**Replicated internal choice**: offers an internal choice over processes $P(i)$ for all i in I

- monitor the time since a state has last been activated, via synchronization with the semantics of State clocks, as required to give semantics to transitions whose guards depend on the time elapsed since the most recent activation of a state.

The State clocks process monitors time since the occurrence of an event *entered*.id(n) for each non-Final state n, and then offers to communicate this time over a channel *get*_id(n). Processes for nodes of type Junction do not synchronize on *entered* events, since they do not have actions or substates, and so become active immediately after synchronizing on *enter*, and also do not have outgoing transitions with deadlines.

To illustrate the role of events modelling the control flow, in Example 2 we reproduce the semantics of the state Prepare of the machine AppleHarvestControl.

Example 2. Semantics of state Prepare.

let

$Inactive \mathrel{\widehat=} SStop \triangle (Activation \Box Termination)$
$Activation \mathrel{\widehat=} enter.\mathrm{id(Prepare)} \rightarrow Active$
$Termination \mathrel{\widehat=} end \rightarrow Skip$
$Active \mathrel{\widehat=} [\![\mathrm{Prepare.entry}]\!]_{\mathscr{A}}^{\mathrm{nops}}; Behaviour; Exiting$
$Behaviour \mathrel{\widehat=} entered.\mathrm{id(Prepare)} \rightarrow During$
$During \mathrel{\widehat=} SStop \triangle interrupt.\mathrm{id(Prepare)} \rightarrow Skip$
$Exiting \mathrel{\widehat=} (SStop \triangle exit \rightarrow Skip); exited \rightarrow Inactive$

within

$Inactive$

It is defined using processes defined in a **let**... **within** block. Initially, Prepare's behaviour is *Inactive*, which behaves as *SStop*, but can be interrupted by *Activation*, via the event *enter*.id(Prepare), or *Termination*, synchronizing on *end* followed by *Skip*. The process $SStop = share \rightarrow SStop$ offers to synchronize indefinitely on the event *share*, used by processes in the semantics to signal and react to changes in the value of shared variables. An *Inactive* state accepts *share* to acknowledge that a shared variable has been changed (although this is not relevant to an inactive state). Once *Active*, the state behaves as $[\![\mathrm{Prepare.entry}]\!]_{\mathscr{A}}^{\mathrm{nops}}$, the process defined by the semantic function $[\![\_]\!]_{\mathscr{A}}^{\mathrm{nops}}$ for actions. Here, the process models the behaviour of Prepare's entry action Prepare.entry. The parameter nops defines the semantics of the operations called by AppleHarvestControl. The process for Prepare's entry action is sequentially composed with *Behaviour*, which concludes the state's activation by synchronizing on *entered*.id(Prepare) and then behaves as *During*.

Because Prepare has no during action, the behaviour of *During* is that of *SStop* with the possibility to be interrupted by *interrupt*.id(Prepare) and terminate. This leads to *Exiting* in the definition of *Active*. Now, *share* is offered until there is a request to *exit*. Next, as Prepare has no exit action, it immediately indicates it has *exited*, and becomes *Inactive*.

In general, once active, a Node n's execution may be interrupted, for example, when an outgoing transition is triggered. This is modelled via synchronization on the event *interrupt*.id(n). In addition, active States may be interrupted by a transition from a parent State. To account for both sources of interruption, in the Control flow process for n's parent NodeContainer p:

1) the event *interrupt*.id(n) is relationally renamed to *interrupt*.id(p), if n is a State;
2) and for each Transition t in p.transitions the event *interrupt*.id(n) is renamed to:

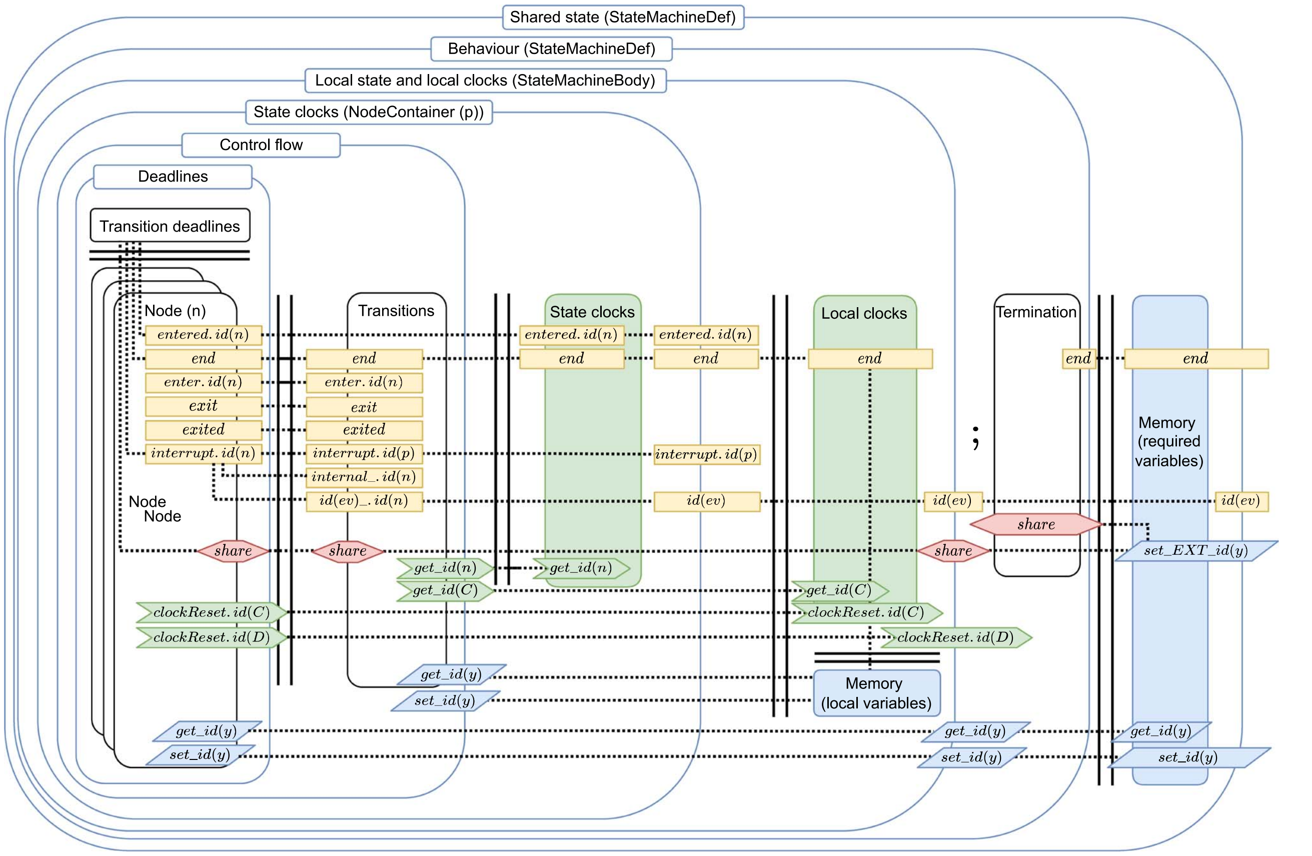


Figure 49. Overall structure of the *tock*-CSP semantics of state machines shown hierarchically, including that of NodeContainer, OperationDef and StateMachineDef. Stacked components and parallel lines indicate parallel composition. Sequential composition is indicated by the semicolon. Bordered boxes indicate visible points of interaction, while dashed lines indicate synchronization between processes on channels. In colour: *yellow* boxes indicate control flow events, *green* boxes are related to clocks, *blue* are related to state variables, and *red* are related to controlling atomic updates to shared state.

a) *internal*.id(n), if t has no trigger;
b) a channel id(ev)_.id(n), if t has a trigger; here ev is the trigger Event t.trigger.event.

The first case (1) allows the process for a composite state to synchronize its interruption with that of an active child State, so that an outgoing transition of the composite state interrupts its machine. Processes for Junctions are not involved in the synchronization, since junctions are decision points and cannot be interrupted.

The second case (2) introduces channels for synchronization with the semantics of Transitions, with two possibilities depending on whether they have a trigger. If a transition t does not have a trigger, its trigger is modelled in the semantics by a channel *internal*. In the process for a NodeContainer (see Fig. 49), *internal* is hidden. This captures the fact that a triggerless Transition can be taken as soon as its guard is *true*. If the transition has a trigger ev, it is represented by a channel id(ev)_.id(n). For every such transition t, in a NodeContainer process, id(ev)_.id(n) is renamed to id(ev), where ev is t.trigger.event and n is t.source, the trigger event and the source State of t (see Fig. 5). With this semantics, events (in triggers) are used for interaction with other components of the system or its environment. Internal control flow, however, is concealed.

The semantics of Transitions accounts for guards by offering, or withholding, synchronization on channels *internal*.id(n) and id(ev)_.id(n), depending on the evaluation of guards. The current value of a variable x, state clock n, or clock C, as used, for example, in a guard, can be queried via synchronization on channels *get*_id(x), *get*_id(n), and *get*_id(C): see Fig. 49.

Variables and clocks are modelled in different ways in the semantics of RoboChart and of open machines. These are presented separately in the sequel.

6.1.2. *RoboChart state machines*

The semantics of a StateMachineDef and of an OperationDef are similar. In both cases, the semantics is defined in terms of the semantic function for a StateMachineBody in general (see Figs 5 and 49). The semantics for an OperationDef, however, is a function that, when applied to arguments corresponding to the parameters of the OperationDef, determines a process that captures the meaning of that OperationDef when called with those arguments. In the case of a StateMachineDef, the semantics does not need to consider parameters, but in addition accounts for termination and required variables. To cater for termination, the semantics of a StateMachineDef defines its contribution to a termination protocol that ensures that it is possible to consider a set of machines running in parallel and sharing a memory, and that guarantees that the set of machines terminates when all its machines terminate. To cater for required variables, the semantics provides a memory process. In contrast, in the semantics of an OperationDef, termination occurs when a final state is reached, so there is no need for a protocol, and the required variables are provided by the calling machine, so there is no need for a memory.

As depicted in Fig. 49 the semantics for a StateMachineDef is defined as the parallel composition of: (a) the sequential composition of a StateMachineBody process with the Termination process, and (b) the memory process for required variables. In the semantics of a StateMachineBody the internal event *end* is used to terminate its behaviour, when a Final state is activated. The process Termination subsequently ($\fatsemi$) offers to synchronize on *end*, so that termination is signalled to other components or the environment.

The semantics of variables and clocks in RoboChart machines is given by a composition with the processes described below, following the hierarchical, component-based nature of RoboChart (see Fig. 49).

- State clocks: as previously discussed, account for the time elapsed since activation of every non-final State $\underline{n}$ in a NodeContainer, with the value available for query via a channel $get\_\underline{id(n)}$;
- Local clocks: account for clocks declared in a state machine. The reset of a Clock $\underline{C}$ is modelled by synchronizing on $clockReset.\underline{id(C)}$, with the time elapsed since then available via a channel $get\_\underline{id(C)}$.
- Memory (local variables and constants): a process that accounts for variables and constants declared in a state machine. Assignment of a new value to a Variable $\underline{v}$ is modelled via a channel $set\_\underline{id(v)}$. The current value is available via a channel $get\_\underline{id(v)}$, and similarly for constants.
- Memory (required variables): similarly, this process accounts for required variables of StateMachineDefs. The difference in this case, however, is that the environment may also provide new values for a Variable $\underline{v}$ via a channel $set\_EXT\_\underline{id(v)}$.

A StateMachineDef must declare all Clocks it directly uses or that are required by operations that it directly calls. Thus, in the process for a StateMachineDef any *clockReset* events are concealed as all Clocks are local.

Next, we illustrate the semantics using the example of the RoboChart machine AppleHarvestControl. What we present is the result of applying the semantic function $[\![\_]\!]_{\mathcal{SMD}}$ to AppleHarvestControl and $\underline{nops}$. That process is presented below, where $\underline{stm}$ stands for AppleHarvestControl. It is defined by a parallel composition ($[\![\ldots]\!]$) of processes synchronizing on *end*, to proceed with termination, and channels for *get*ting, and *set*ting the value of the required variables apples and positions. Data is captured by a process defined by a metafunction $\underline{\text{sharedVarMemory(stm)}}$. The definition of this and other functions to follow can be found in [48].

Example 3. *tock*-CSP semantics of the RoboChart state machine AppleHarvestControl.

$$[\![\text{AppleHarvestControl}]\!]^{nops}_{\mathcal{SMD}} =$$

$$\left(\begin{array}{l} \left([\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}} ; SStop \triangle end \to Skip\right) \\ \left[\!\!\left[x : \left\{ \begin{array}{l} set\_EXT\_\underline{id(apples)}, \\ set\_EXT\_\underline{id(positions)} \end{array} \right\} \bullet share \leftarrow x \right]\!\!\right] \\ \quad [\![\{\!|share|\!\}]\!] \\ Skip \end{array}\right)$$

$$\left[\!\!\left[\left\{ \begin{array}{l} end, \\ get\_\underline{id(apples)}, set\_\underline{id(apples)}, \\ set\_EXT\_\underline{id(apples)}, \\ get\_\underline{id(positions)}, set\_\underline{id(positions)}, \\ set\_EXT\_\underline{id(positions)} \end{array} \right\} \right]\!\!\right]$$

$$\underline{\text{sharedVarMemory(stm)}}$$

The semantics of a required variable, for example, apples, can be modelled by the parametric and recursive memory process M sketched below.

$$M(a) = \left(\begin{array}{l} get\_\underline{id(apples)}!\,a \to M(a) \\ \Box set\_\underline{id(apples)}?x \to M(x) \\ \Box set\_EXT\_\underline{id(apples)}?x \to M(x) \end{array} \right)$$

For a complete definition, an initial value is provided to reflect the default for the type of the variable being modelled. M offers, in an external choice, the possibility to get the current value a via $get\_\underline{id(apples)}$, or set a new value x, via $set\_\underline{id(apples)}$ or $set\_EXT\_\underline{id(apples)}$, with the behaviour given by the recursion on M with the chosen value x as a parameter.

Composed in parallel with $\underline{\text{sharedVarMemory(stm)}}$, in Example 3, is the process semantics of $\underline{\text{stm}}$'s behaviour. At the core is the process $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$ defined by the semantic function for a StateMachineBody. A renaming ($[\![\ldots]\!]$) maps the event *share* ($\leftarrow$) to $set\_EXT\_\underline{id(apples)}$ and $set\_EXT\_\underline{id(positions)}$, to allow external updates to the values of the variables apples and positions to be observed as *share* events by the process $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$. If a state machine has no required variables, and thus the renaming relation is empty, the event *share* is not renamed. So, we have a parallel composition with *Skip* synchronizing on *share*. Since *Skip* terminates and does not agree to engage in *share* events, they are blocked, if they are not renamed.

The behaviour of $\underline{\text{stm}}$ is defined by the sequential composition of $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$ and $SStop \triangle end \to Skip$. The latter defines the behaviour when $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$ has terminated and allows the value of required variables to be updated via synchronization on *set_EXT* channels while waiting to agree on termination via *end*. AppleHarvestControl has no final state, and so $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$ never terminates. Next, we sketch $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$.

Example 4. Sketch of $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$.

$$\left(\begin{array}{l} \left([\![\text{stm}]\!]^{\text{nops}}_{\mathcal{NC}} [\![\{\!|interrupt|\!\}]\!] Skip\right) \setminus \{\!|entered|\!\} \\ \left[\!\!\left[\left\{ \begin{array}{l} end, \\ get\_\underline{id(img)}, set\_\underline{id(img)}, \\ get\_\underline{id(localized)}, set\_\underline{id(localized)}, \\ get\_\underline{id(nextApple)}, set\_\underline{id(nextApple)}, \ldots \end{array} \right\} \right]\!\!\right] \\ \left(\begin{array}{l} \underline{\text{varMemory(stm)}} \\ \quad [\![\{end\}]\!] \\ \underline{\text{constMemory(stm)}} \\ \quad [\![\{end\}]\!] \\ \underline{\text{clocks(stm.clocks)}} \end{array} \right) \end{array}\right)$$

$$\setminus \left\{ \begin{array}{l} end, get\_\underline{id(img)}, set\_\underline{id(img)}, \\ get\_\underline{id(localized)}, set\_\underline{id(localized)}, \\ get\_\underline{id(nextApple)}, set\_\underline{id(nextApple)}, \ldots \end{array} \right\}$$

The semantics defined by $[\![\text{stm}]\!]^{\text{nops}}_{\mathcal{SMB}}$ is reproduced in Example 4. The behaviour is given in terms of the NodeContainer semantics of $\underline{\text{stm}}$, defined by another semantic function $[\![\_]\!]^{\text{nops}}_{\mathcal{NC}}$, which we discuss below. It is constrained, first of all, by a parallel composition with *Skip*, synchronizing on the channel *interrupt* to block it. We recall that, as discussed in subsubsection 6.1.1, the channel *interrupt* can be used by composite states to synchronize their interruption with their children. A state machine, however, is not interruptible, so this parallel composition makes *interrupt* unavailable.

The channel *entered* is hidden, given that entering of states is a control flow mechanism, and thus is not visible in the semantics of a state machine. This behaviour is then composed in parallel with processes defined by application of the following metafunctions:

- varMemory(stm), which defines a process similar to sharedVarMemory(stm), models variables defined locally in AppleHarvestControl, synchronizing on channels for *get*ting and *set*ting the value of the variables img, localized and nextApple.
- constMemory(stm), which models local constants, synchronizing on channels for *get*ing the value of constants defined by the interfaces TimeConstants and Locations. For brevity we elide them from the channel sets shown in Example 4.
- clocks(stm.clocks), which models local clocks. The state machine AppleHarvestControl has no clocks, so synchronization on *end* is the only behaviour.

Finally, the event *end* controlling termination and the channels for *get*ting and *set*ting the value of local variables and constants are hidden.

NodeContainer The NodeContainer semantics of the machine AppleHarvestControl is sketched in Example 5.

Example 5. Sketch of the semantics of $\underline{[\![\mathsf{stm}]\!]^{\mathsf{nops}}_{\mathcal{NC}}}$.

$$\left(\begin{array}{l}\left(\begin{array}{l}\underline{\mathsf{controlFlow}(\mathsf{stm})^{\mathsf{nops}}}\\ \setminus \{\!|enter, exit, exited, internal|\!\}\end{array}\right)\\ \left[\!\left[\begin{array}{l} takePic\_.\underline{\mathsf{id}(\mathsf{Prepare})} \leftarrow takePic,\\ endGoHome\_.\underline{\mathsf{id}(\mathsf{GoingHome})} \leftarrow endGoHome\end{array}\right]\!\right]\\ \quad [\![\{\!| end, entered.\underline{\mathsf{id}(\mathsf{Prepare})}, get\_\underline{\mathsf{id}(\mathsf{Prepare})}, \ldots |\!\}]\!]\\ \underline{\mathsf{clocks}(\{\mathsf{Prepare}, \ldots\})}\end{array}\right)$$

$$\setminus \{\!| get\_\underline{\mathsf{id}(\mathsf{Prepare})}, \ldots |\!\}$$

The behaviour is given, first of all, in terms of another metafunction $\underline{\mathsf{controlFlow}(\mathsf{stm})^{\mathsf{nops}}}$, discussed next. The channels *enter*, *exit*, *exited* and *internal* are hidden, and channels modelling transition triggers are renamed to conceal the identifiers of states: *takePic*_id(Prepare), which accounts for the transition between Prepare and LocalizeFruit, is renamed to *takePic*; and *endGoHome*_id(GoingHome), for the transition between GoingHome and endGoHome, is renamed to *endGoHome*. This process is then composed in parallel with that defined by application of the function clocks(...), synchronizing on *end* and channels *entered*.id(n) and *get*_id(n) for each non-final State n in AppleHarvestControl. This allows the clocks process to be terminated (*end*), and exchange of relevant information related to time conditions: entering a state is signalled to the clocks process, which records time and can provide the time since entering a state via *get*_id(n) channels. In Example 5 the complete channel set is elided, with only *end* and the channels related to the state Prepare shown. The metafunction clocks, defined here over a set of States, is applied to the set of non-final states of AppleHarvestControl, which includes Prepare. Finally, the channels for *get*ing the time since entering a state, such as *get*_id(Prepare), are hidden.

Control Flow The interaction between the Nodes and Transitions of AppleHarvestControl is captured by the semantics sketched in Example 6.

Example 6. Semantics of control flow of stm.

$$\left(\begin{array}{l}\left(\begin{array}{l}\underline{\mathsf{composeTimedNodes}(\mathsf{stm})^{\mathsf{nops}}}\\ \left[\!\left[\begin{array}{l} interrupt.\underline{\mathsf{id}(\mathsf{i_0})} \leftarrow internal.\underline{\mathsf{id}(\mathsf{i_0})},\\ interrupt.\underline{\mathsf{id}(\mathsf{Prepare})} \leftarrow takePic\_.\underline{\mathsf{id}(\mathsf{Prepare})},\\ interrupt.\underline{\mathsf{id}(\mathsf{Prepare})} \leftarrow interrupt.\underline{\mathsf{id}(\mathsf{stm})},\\ set\_\underline{\mathsf{id}(\mathsf{positions})} \leftarrow setL\_\underline{\mathsf{id}(\mathsf{positions})}, \ldots \end{array}\right]\!\right]\end{array}\right)\\ \quad [\![\underline{\mathsf{cs}} \cup \{\!| share, setL\_\underline{\mathsf{id}(\mathsf{positions})}, \ldots |\!\}]\!]\\ \left(\begin{array}{l}(enter.\underline{\mathsf{id}(\mathsf{i_0})} \rightarrow \underline{\mathsf{transitions}(\mathsf{stm})^{\mathsf{nops}}})\\ \left[\!\left[\begin{array}{l} share \leftarrow share,\\ share \leftarrow setL\_\underline{\mathsf{id}(\mathsf{positions})}, \ldots\end{array}\right]\!\right]\end{array}\right)\end{array}\right)$$

$$\left[\!\left[\; setL\_\underline{\mathsf{id}(\mathsf{positions})} \leftarrow set\_\underline{\mathsf{id}(\mathsf{positions})}, \ldots \;\right]\!\right]$$

where

$$\underline{\mathsf{cs}} = \left\{\!\!\left|\begin{array}{l} end, enter, exit, exited, interrupt,\\ internal.\underline{\mathsf{id}(\mathsf{i_0})}, takePic\_.\underline{\mathsf{id}(\mathsf{Prepare})}, \ldots\end{array}\right|\!\!\right\}$$

The behaviour is defined by the parallel composition of the semantics of its Nodes, defined by application of a function composeTimedNodes with channels *interrupt* renamed for each transition (as discussed in subsubsection 6.1.1), and the semantics of its transitions. There, the prefixing on *enter*.$\underline{\mathsf{id}(\mathsf{i_0})}$ requests initialization of AppleHarvestControl's Initial junction $\underline{\mathsf{i_0}}$ before proceeding further as defined by the metafunction application $\underline{\mathsf{transitions}(\mathsf{stm})^{\mathsf{nops}}}$. Assignments in the semantics of Nodes to Variables used to guard transitions need to be synchronized with the semantics of the transitions to ensure the sequential control flow is consistent. For example, the assignment to the variable positions in the state GetApple may affect the evaluation of the guard on the transition to GoingHome, so the processes must synchronize on *set*_id(positions). On the other hand, an assignment on a transition, such as that between Prepare and LocalizeFruit, must happen independently, that is, without synchronization. To define this protocol, we use renaming. In the above example, writes from Nodes to positions take place via *set*_id(positions). This channel is, however, renamed to a *setL*_id(positions), which is used to synchronize with the semantics of transitions by renaming *share* both to itself and to *setL*_id(positions). Thus, any assignment to positions by a Node is seen by the process transitions as a synchronization on *share*. The new value can be queried via *get*_id(positions). Finally, after the parallel composition, *setL*_id(positions) is renamed to *set*_id(positions) so that other processes can observe the assignments as before. In the example, the renaming and synchronization with *share* involving other variables are elided. The parallel composition also requires synchronization on the channel set cs, comprising events *end*, *enter*, *exit*, *exited* and *interrupt* related to control flow, and channels related to triggers of transitions, such as *internal*.$\underline{\mathsf{id}(\mathsf{i_0})}$ and *takePic*_.id(Prepare). This ensures consistent flow evolution.

This concludes the discussion of the semantics of RoboChart machines. Next, we discuss open machines.

6.1.3. *Open machines*

In Example 7 we reproduce the semantics of the open state machine depicted in subsubsection 38, of type CompOpenStateMachine (see subsection 13), referred to as ostm, which is itself composed of two open machines, ostm.left, that contains an EState named Done with a single transition to a NodeNameRef named f0, and $\underline{\mathsf{ostm.right}}$, containing an EFinal state named f0.

Example 7. decompBOM(LocalizeFruit): semantics for the model shown in subsubsection 38.

The semantics is defined by the parallel composition of the semantics of $\underline{\mathsf{ostm.left}}$, obtained by application of a semantic function $\underline{[\![\_]\!]^{\mathsf{nops}}_{\mathscr{BOSM}}}$, that we explain in the sequel, and that of $\underline{\mathsf{ostm.right}}$, synchronizing on channels *end*, r_{enter}, and *share*. The new channel r_{enter} allows the processes for $\underline{\mathsf{ostm.left}}$ and $\underline{\mathsf{ostm.right}}$ to synchronize internally on the activation requests for their shared states, while keeping the actual control flow channel *enter* available for the processes for other machines.

$$\left(\begin{array}{l}\left(\begin{array}{l}\underline{[\![\mathsf{ostm.left}]\!]^{\mathsf{nops}}_{\mathscr{BOSM}}}\\ [\![\; enter.\underline{\mathsf{id}(\mathsf{f}_0)} \leftarrow r_{enter}.\underline{\mathsf{id}(\mathsf{f}_0)} \;]\!]\end{array}\right)\\ \mathrel{[\![\{\!|end, r_{enter}, share|\!\}]\!]}\\ \left(\begin{array}{l}\underline{[\![\mathsf{ostm.right}]\!]^{nops}_{\mathscr{BOSM}}}\\ \left[\!\!\left[\begin{array}{l} enter.\underline{\mathsf{id}(f_0)} \leftarrow enter.\underline{\mathsf{id}(f_0)},\\ enter.\underline{\mathsf{id}(f_0)} \leftarrow r_{enter}.\underline{\mathsf{id}(f_0)}\end{array}\right]\!\!\right]\end{array}\right)\end{array}\right) \setminus \{\!|r_{enter}|\!\}$$

For example, in $\underline{\mathsf{ostm.right}}$, the entry point f0 coincides in name with that of a NodeNameRef in $\underline{\mathsf{ostm.left}}$ that is the target of a transition from Done. Thus, the event $enter.\underline{\mathsf{id}(\mathsf{f}_0)}$ is renamed to $r_{enter}.\underline{\mathsf{id}(\mathsf{f}_0)}$ in the process for $\underline{\mathsf{ostm.left}}$, and in the process for $\underline{\mathsf{ostm.right}}$ it is renamed to: (a) $r_{enter}.\underline{\mathsf{id}(\mathsf{f}_0)}$, so that the transition to the NodeNameRef can request activation of f0; (b) and to $enter.\underline{\mathsf{id}(\mathsf{f}_0)}$, so that composition with other open machines whose transitions may also target f0 is feasible by allowing independent synchronization with $enter.\underline{\mathsf{id}(\mathsf{f}_0)}$. The new event r_{enter} is hidden.

The semantics of $\underline{\mathsf{ostm.left}}$, which, we recall, is a BasicOpenStateMachine, is sketched in Example 8. In this case, the semantics is not given by a composition with a memory process, as illustrated by Example 3. The semantics of an open machine is closer to that of a State, illustrated by Example 2, and is also defined using a **l**et…**w**ithin block. For a machine, however, instead of an *Activation* process, we have just the *Active* process, because a machine does not need to report on the status of its activation. An active machine also has no (entry) actions, so *Active* is simpler. Similarly, a machine has no during action and no need to report on its exiting. Finally, the *Behaviour* of the machine is triggered by an *enter*, rather than *entered*, channel.

Example 8. Semantics of the left-hand machine in decompBOM(LocalizeFruit) shown in subsubsection 38.

let

$$\begin{array}{l} Inactive = SStop \triangle (Termination \,\Box\, Active)\\ Termination = end \rightarrow Skip\\ Active = Behaviour; Inactive\\ Behaviour = enter?i : \{\underline{\mathsf{id}(\mathsf{Done})}\} \rightarrow\\ \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\underline{\mathsf{composeTimedNodes}(\mathsf{stm})^{\mathsf{nops}}}\\ \left[\!\!\left[\begin{array}{l} interrupt.\underline{\mathsf{id}(\mathsf{Done})} \leftarrow internal.\underline{\mathsf{id}(\mathsf{Done})},\\ interrupt.\underline{\mathsf{id}(\mathsf{Done})} \leftarrow interrupt.\underline{\mathsf{id}(\mathsf{stm})}\end{array}\right]\!\!\right]\end{array}\right)\\ \left[\!\!\left[\left\{\begin{array}{l} share, end, enter, exit,\\ exited, interrupt, internal.\underline{\mathsf{id}(\mathsf{t}_0)}\end{array}\right\}\right]\!\!\right]\\ (enter.i \rightarrow \underline{\mathsf{transitions}(\mathsf{stm})^{\mathsf{nops}}})\end{array}\right)\\ \setminus (\{\!|enter, exit, exited, internal|\!\} \setminus \{\!|enter.\underline{\mathsf{id}(\mathsf{f}_0)}|\!\})\end{array}\right)\\ \setminus \{end\}\end{array}$$

within

$Inactive$

In *Active*, it offers to synchronize on *enter* with a parameter *i* drawn from the set of identifiers of the machine's entry points. In this example, the set is a singleton, $\underline{\mathsf{id}(\mathsf{Done})}$, since there is only one entry point in $\underline{\mathsf{ostm.left}}$. The value of *i* is then subsequently used to require activation of the state Done, similarly to the control flow semantics illustrated in Example 6. The parallel composition and the renaming over $\underline{\mathsf{composeTimedNodes}}$ are similar to those in the control flow semantics of a RoboChart machine (see Example 6). The subsequent hiding of channels *enter*, *exit*, *exited*, and *internal* is similar to that of the NodeContainer semantics (as illustrated in Example 5), which uses the control flow semantics. The exception here is that events *enter* pertaining to nodes referenced by NodeNameRefs are excluded from the hiding, and so the event $enter.\underline{\mathsf{id}(\mathsf{f}_0)}$ is not hidden. This allows the composition with the process for another machine, namely $\underline{\mathsf{ostm.right}}$ in Example 7, to reflect the sequential control flow, where, for example, the entry point f0 in $\underline{\mathsf{ostm.right}}$ is activated by the transition out of the state Done in $\underline{\mathsf{ostm.left}}$. Similarly to a Final state, once a NodeNameRef is activated, it can be terminated via synchronization on *end*, which leads to the termination of *Behaviour*. As for RoboChart machines (see Example 3), *end* is hidden. Afterwards, the open machine becomes *Inactive*.

As said previously, equality of open machines is parametrised by the OperationDefinitions for the operations that can be called in these machines. Formally, $OM_1 =_{Op} OM_2$ is defined as follows. Here, $\underline{[\![\_]\!]^{\mathsf{nops}}_{\mathscr{OSM}}}$ is the semantic function for OpenStateMachines, which is defined in terms of those for BasicOpenStateMachines and for CompOpenStateMachines, and like those, takes a parameter that defines the semantics of operations. Below, that parameter is the set defined by $\underline{\mathsf{nops}}$, including the semantics of the operations other than *Op*, and the set containing just the pair $\underline{\mathsf{Op.name} \mapsto [\![\mathsf{Op}]\!]^{\mathsf{nops}}_{\mathscr{OPD}}}$ that gives the semantics of *Op* as defined by $\underline{[\![\_]\!]^{-}_{\mathscr{OPD}}}$.

$$OM_1 =_{Op} OM_2 \iff \left(\begin{array}{l}\underline{[\![\mathsf{OM}_1]\!]^{\mathsf{nops}\cup\left\{\mathsf{Op.name}\mapsto[\![\mathsf{Op}]\!]^{\mathsf{nops}}_{\mathscr{OPD}}\right\}}_{\mathscr{OSM}}}\\ =\\ \underline{[\![\mathsf{OM}_2]\!]^{\mathsf{nops}\cup\left\{\mathsf{Op.name}\mapsto[\![\mathsf{Op}]\!]^{\mathsf{nops}}_{\mathscr{OPD}}\right\}}_{\mathscr{OSM}}}\end{array}\right)$$

The open machines OM_1 and OM_2 are related by $=_{Op}$ exactly when their semantics are equal, as given by $\underline{[\![\_]\!]^{-}_{\mathscr{OSM}}}$ with the parameter as defined above.

This concludes the discussion of the semantics of open machines. Next, we use the semantics to justify the soundness of the normalization laws.

6.2. Proofs of soundness

In what follows, we justify the soundness of selected Laws 2 and 6. First we formalize them, and then sketch our proofs of soundness using the semantics. Results obtained as part of proving Law 2 are useful to prove soundness of Laws 3, 15, and 16 following a similar proof strategy. Similarly, results established for Law 6 are relevant to justify that Laws 11 and 17 are sound. Complete proofs can be found in [37].

6.2.1. *Law 2* (*intro-call-for-act-state*)

Before presenting the formalization of Law 2, we first define a relation that allows comparing state machines.

Relating structurally similar NodeContainers We recall that both state machines and states are NodeContainers. The relation $(nc1, nc2) \approx^{\mathrm{N}}_{\mathscr{NC}} (nc3, nc4)$ allows us to compare a machine $nc1$ with a substate, at any level, $nc2$, to another machine $nc3$, whose

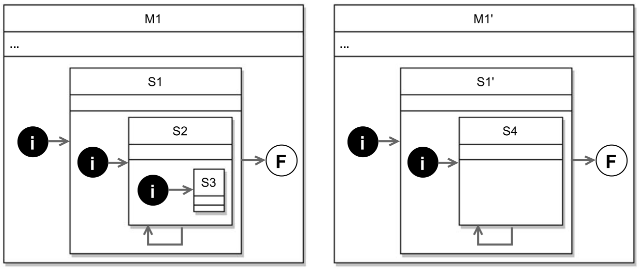


Figure 50. Example of the use of the relation $\approx^{N}_{\mathcal{NC}}$ between pairs (M1, S2) and (M1′, S4). Here, it characterizes that M1 and M1′ are state machines that differ in that the state S2 of M1 is replaced with S4 in M1′.

only difference is in that *nc*2 is replaced with *nc*4 in *nc*3. Informally, $(nc1, nc2) \approx^{N}_{\mathcal{NC}} (nc3, nc4)$ if, and only if, (a) *nc*2 is equal to *nc*1 or is one of its nodes (possibly indirectly, inside a composite state of *nc*1, for instance); and (b) *nc*3 differs from *nc*1 just in that *nc*2 is replaced with *nc*4 in *nc*3.

For the simple case, in which we want to replace a whole machine with another, we do not require this relation. Equally, if *nc*2 is a direct state of *nc*1, then $(nc1, nc2) \approx^{N}_{\mathcal{NC}} (nc3, nc4)$ holds when

$$nc3 = nc1 \oplus \{\ nodes \mapsto (nc_1.nodes \setminus \{nc2\}) \cup \{nc4\},\ trans \mapsto \phi(nc2, nc4)(\!| nc1.trans |\!)\}$$

where the *transitions* of *nc*1 that have *nc*2 as source or target are changed to consider *nc*4 via the relational image ($(\!|\,|\!)$) through ϕ defined below. Here, we use the overriding operator ($\oplus$) to change the values of the attributes of an element of the metamodel, above *nc*1, taking into account the pairs in a set: above, two pairs giving new values to the attributes *nodes* and *trans*.

Below, ϕ is applied to NodeContainers *nc*5 and *nc*6 to define a function from transitions to transitions. Here $\phi(nc5, nc6)$ is a lambda function that applies to a transition *t*; if the *source* or *target* of *t* is *nc*5, then it is replaced by *nc*6. Otherwise, no pairs are added to the overriding set. Transitions may have a *condition* that depends on the time since a state has been entered, so we apply ϕ_{exp}, whose definition is omitted, to replace occurrences of *nc*5 by *nc*6 also in that *condition*.

Definition 6.1.

$$\phi(nc5, nc6) =$$
$$\lambda t \bullet t \oplus \left\{ \begin{array}{l} \textbf{if } t.source = nc5 \textbf{ then } source \mapsto nc6, \\ \textbf{if } t.target = nc5 \textbf{ then } target \mapsto nc6, \\ cond \mapsto \phi_{exp}(t.cond, nc5, nc6) \end{array} \right\}$$

The more general relation $(nc1, nc2) \approx^{N}_{\mathcal{NC}} (nc3, nc4)$ captures also situations in which *nc*2 is deeply embedded in the structure of *nc*1. For example, in Fig. 50, we relate a state S2 within the machine M1, and a state S4 within M1′ using $(M1, S2) \approx^{N}_{\mathcal{NC}} (M1', S4)$. The machines M1 and M1′ are identical, except that S1 is replaced by S1′, where S2 is replaced by S4. The transitions of M1 that concern S1 instead refer to S1′ in M1′, and similarly those of S1 that concern S2 instead refer to S4 in S1′. Similarly, (S1, S2) is also related to (S1′, S4) by $\approx^{N}_{\mathcal{NC}}$. Its formal definition is below.

Definition 6.2.

$$(m, s) \approx^{N}_{\mathcal{NC}} (m', s') \Leftrightarrow (m = s \wedge m' = s') \vee$$
$$\exists nc, nc' : NodeContainer \bullet$$
$$nc \in m.nodes \wedge \#m.nodes = \#m'.nodes \wedge$$
$$m' = m \oplus \left\{ \begin{array}{l} nodes \mapsto (m.nodes \setminus \{nc\}) \cup \{nc'\} \\ trans \mapsto \phi(nc, nc')(\!| m.trans |\!) \end{array} \right\} \wedge$$
$$(nc, s) \approx^{N}_{\mathcal{NC}} (nc', s')$$

It is defined over pairs of NodeContainers, such that:

1) (s, s) is related to (s', s') for arbitrary s and s'; or
2) (m, s) is related to (m', s') if, and only if,
   a) there are NodeContainers *nc* and *nc*′, direct descendants of *m* and *m*′, given that we require (i) $nc \in m.nodes$, (ii) the nodes $m'.nodes$ of m' can be characterized by adding nc' after removal of *nc*; and (iii) the number of nodes in *m* and *m*′ are the same ($\#m.nodes = \#m'.nodes$), so that *nc*′ is not already in *m*;
   b) the transitions of *m*′ can be characterized from those of *m* using the function ϕ; and
   c) (nc, s) is related to (nc', s') by $\approx^{N}_{\mathcal{NC}}$.

Next, we use this relation in the formalization of Law 2. There, the equality is stated over the semantics of StateMachineDefs $\underline{m}$ and $\underline{m'}$ as given by the semantic function $[\![\_]\!]^{-}_{\mathcal{SMD}}$, which, as explained in the previous section, is parametrised by the semantics of operations: $\underline{\mathbf{nops}_1}$ and $\underline{\mathbf{nops}_2}$ above, which are partial functions ($\mapsto$) associating the $\underline{\text{Name}}$ of an operation with its CSP process semantics. Afterwards, we have the declaration of metavariables used in Law 2 giving their types. Namely, we declare $\underline{\text{S}}$, $\underline{\text{s}}$, and $\underline{\text{IOpS}}$, and $\underline{\text{action}_\text{s}}$, which capture model elements of the state machines. In the diagram for Law 2, we use action s to refer to an action of $\underline{\text{S}}$; above, we use $\underline{\text{action}_\text{s}}$ instead to represent that action. We also use $\underline{\text{OpS}(...)}$ in the diagrammatic version of Law 2 to refer both to a call to an operation and to the signature used in its definition. We here declare $\underline{\text{call}_\text{OpS}}$ to represent the call, and $\underline{\text{OpS}}$ to represent the operation itself.

Formalisation 1. (Law 2).

$$\underline{[\![\text{m}: \texttt{StateMachineDef}]\!]^{\text{nops}_1}_{\mathcal{SMD}} = [\![\text{m}': \texttt{StateMachineDef}]\!]^{\text{nops}_2}_{\mathcal{SMD}}}$$

where

- $\underline{\text{S}:\text{State}}$, $\underline{\text{s}:\text{Statement}}$, $\underline{\text{IOpS}:\text{Interface}}$,
- $\underline{\text{action}_\text{s} : \text{Action}}$,
- $\underline{\text{call}_\text{OpS}:\text{Call}}$, $\underline{\text{OpS} : \text{OperationDef}}$

1. $\underline{\text{action}_\text{s} \in \text{S.actions} \wedge \text{s} = \text{action}_\text{s}.\text{action}}$
2. $\underline{\text{OpS} = \text{actionOperation(s)}}$
3. $\underline{\text{call}_\text{OpS}.\text{op} = \text{opSig(OpS)} \wedge \text{call}_\text{OpS}.\text{args} = \text{null}}$
4. $\underline{\text{IOpS} = \{\text{operations} \mapsto \{\text{opSig(OpS)}\}\}}$
5. $\underline{\text{nops}_2 = \text{nops}_1 \cup \{\text{OpS.name} \mapsto [\![\text{OpS}]\!]^{\text{nops}_1}_{\mathcal{OPD}}\}}$
6. $\underline{(\text{m} \oplus \{\text{RInterfaces} \mapsto \text{m.RInterfaces} \cup \text{IOpS}\}, \text{S}) \approx^{N}_{\mathcal{NC}} \left(\text{m}', \text{S} \oplus \left\{ \begin{array}{l} \text{actions} \mapsto \\ \quad (\text{S.actions} \setminus \{\text{action}_\text{s}\}) \\ \quad \cup \{\text{action}_\text{s} \oplus \{\text{action} \mapsto \text{call}_\text{OpS}\}\} \end{array} \right\}\right)}$ ■

Following the declarations, a list of numbered properties restrict the values of the metavariables to capture the assumptions in the diagram for Law 2. First, in A1, we record the fact that Law 2 uses $action_s$ as an Action of a State S and that s is the statement of $action_s$. We recall that the statement of an action is recorded via an attribute named action according to the metamodel in Fig. 5. A2 formalizes the definition of OpS using a metafunction actionOperation(s), defined in [37]; as the name suggests, it defines an action operation for the given statement s, using interfaces req's and defEvents's of Law 2. [A3] records the Call's target as the operation OpS and that there are no arguments. [A4] formalizes the definition of the interface IOpS: its component operations maps to the singleton set containing the signature of OpS, obtained by a projection function opSig. [A5] states that the parameter $nops_2$ is defined by extending $nops_1$ with the semantics of OpS. Finally, A6 captures how the machine m′ is defined in terms of m, after it is enriched with the required interface IOpS: the only difference in m′ is that S has the original $action_s$ removed and a new action added, so that the action attribute is replaced by $call_{Ops}$.

Law 2 and all the other laws for RoboChart machines apply to both StateMachineDefs and OperationDefs; in other words, they are laws of StateMachineBody. The formalization of Law 2 when m and m′ are OperationDefs is very similar to that presented above. There are only two differences: the equality is for the semantics defined by the function $[\![\_]\!]_{\mathscr{OPD}}$, instead of $[\![\_]\!]_{\mathscr{SMD}}$, and [A3] records that the arguments to be passed in the Call are given by a function usedArgs(s), which, as explained before, if m is an OperationDef, captures the parameters of m used in s, if any.

Accordingly, the proof of Law 2 has two cases: one for StateMachineDefs and another for OperationDefs. We sketch below the proof for StateMachineDefs. For OperationDefs, the proof strategy is similar and simpler.

Proof. (Case StateMachineDefs m and m′) We define an action $a_0 = action_s \oplus \{action \mapsto call_{Ops}\}$, a new state $S_0 = S \oplus \{actions \mapsto (S.actions \setminus \{action_s\}) \cup \{a_0\}$ that adds to S the action a_0, and a machine that extends m by including IOpS as an additional required interface $m_0 = m \oplus \{RInterfaces \mapsto m.RInterfaces \cup IOpS\}$, for which we can establish the following results.

R1 $(m_0, S) \approx^{N}_{\mathscr{NC}} (m', S_0)$;

R2 $vars(S_0) = vars(S)$, that is, the set of variables used in S and S_0 is the same;

R3 $usedOps(a_0) = usedOps(action_s) \cup \{OpS\}$, since a_0 calls OpS, and in the definition of OpS, we have s. So, $usedOps(S_0) = usedOps(S) \cup \{OpS\}$, that is, the set of operations used by S_0, those called by actions of S_0, its substates or transitions, is that used by S augmented with OpS, thus we also have that $usedOps(m') = usedOps(m_0) \cup \{OpS\}$, and $usedOps(m') \cap usedOps(m_0) = usedOps(m_0)$, that is, the operations used by both m′ and m_0 are the same. Therefore, they behave the same:

$$\forall op : usedOps(m') \cap usedOps(m_0) \bullet$$
$$[\![nops_1(op)]\!]^{nops_1}_{\mathscr{OPD}} = [\![nops_2(op)]\!]^{nops_2}_{\mathscr{OPD}}$$

R4 $[\![S]\!]^{nops_1}_{\mathscr{N}} = [\![S_0]\!]^{nops_2}_{\mathscr{N}}$, that is, the semantics of the state S_0 is the same as that of S.

■

We recall that the proofs of these results and all others omitted here are in [37]. They justify the application of the two key lemmas below. Lemma 6.1 establishes that requiring additional operations has no impact on the semantics of a state machine.

Lemma 6.1. [Augment required operations]

$$[\![m : StateMachineDef]\!]^{nops_1}_{\mathscr{SMD}}$$
$$=$$
$$[\![m \oplus \{RInterfaces \mapsto m.RInterfaces \cup RIopS\}]\!]^{nops_1}_{\mathscr{SMD}}$$

Proof. Using the definition of $[\![\_]\!]_{\mathscr{SMD}}$, which does not rely on the definition of required interfaces to give semantics to operation calls. Instead, we recall, the semantics of operations is passed as a parameter. Here, the argument $nops_1$ is used on both sides of the equation, thus, augmenting the required interfaces of m with RIopS has no impact on the behaviour. ■

Lemma 6.2 below can be seen as a more general account of Law 2 in terms of the semantics of states. It establishes that the semantics of two state machines m and m′ is the same when $(m, s) \approx^{N}_{\mathscr{NC}} (m', s')$ and the semantics of states s and s′ is also the same. Similarly to A6 in the formalization of Law 2, here P1 requires that (m, s) is related to (m′, s′) by $\approx^{N}_{\mathscr{NC}}$, where s : State constrains the type of s to be a State. Proviso P2 requires that the variables used by s and s′ are the same, and P3 requires that operations used by both machines have the same behaviour. Proviso P4, in addition, states that the semantics of s′ and s, given by a function $[\![\_]\!]_{\mathscr{N}}$ and parametrised by the semantics of operations $nops_1$ and $nops_2$, respectively, must be equal.

Lemma 6.2.

$$[\![m : StateMachineDef]\!]^{nops_1}_{\mathscr{SMD}} = [\![m' : StateMachineDef]\!]^{nops_2}_{\mathscr{SMD}}$$

provided:

P1 $(m, s : State) \approx^{N}_{\mathscr{NC}} (m', s' : State)$

P2 $vars(s) = vars(s')$

P3 $\left(\begin{array}{l}\forall op : usedOps(m') \cap usedOps(m) \bullet \\ [\![nops_1(op)]\!]^{nops_1}_{\mathscr{OPD}} = [\![nops_2(op)]\!]^{nops_2}_{\mathscr{OPD}}\end{array}\right)$

P4 $[\![s : State]\!]^{nops_1}_{\mathscr{N}} = [\![s' : State]\!]^{nops_2}_{\mathscr{N}}$

Proof. There are two cases to consider:

1) $s \in m.nodes$ and $s' \in m'.nodes$: so we can infer from Proviso P1 that m.nodes and m′.nodes differ only in that s is replaced by s′ in m′.nodes. Proviso P4 requires that the behaviour of the states s′ and s is the same, so they engage in the same control flow events *enter*.id(s), *entered*.id(s), and *interrupt*.id(s), and so id(s) = id(s′) holds. Moreover, we can infer that the semantics of Transitions involving these states is the same. We recall that the control flow semantics of NodeContainers is defined by a parallel composition of the semantics of nodes and transitions, taking into account any variables shared between them. Proviso P2 ensures that the variables used by s and s′ are the same. From this and Proviso P3 we can infer that the control flow semantics of m and m′ match, as do their NodeContainer semantics given by $[\![\_]\!]_{\mathscr{NC}}$. We recall that the semantics of a StateMachineDef, as given by $[\![\_]\!]_{\mathscr{SMD}}$, is defined in terms of $[\![\_]\!]_{\mathscr{NC}}$ in a parallel composition with a model of its variables and clocks. From

Proviso P1 the variables and clocks declared by $\underline{m}$ and $\underline{m'}$ are the same, and so $\underline{[\![m]\!]^{nops_1}_{\mathscr{SMD}}} = \underline{[\![m']\!]^{nops_2}_{\mathscr{SMD}}}$ as required.

2) $\underline{s}$ and $\underline{s'}$ are not direct descendants of $\underline{m}$ and $\underline{m'}$, so from Proviso P1 and the definition of $\approx^{N}_{\mathscr{NC}}$ we can infer that there exist substates $\underline{z}$ and $\underline{z'}$ of $\underline{m}$ and $\underline{m'}$, respectively, such that $\underline{(z, s)} \approx^{N}_{\mathscr{NC}} \underline{(z', s')}$. We can show that $\underline{[\![z]\!]^{nops_1}_{\mathscr{N}}} = \underline{[\![z']\!]^{nops_2}_{\mathscr{N}}}$. For that, we consider the semantics of composite states, which is defined in terms of their semantics as NodeContainers, that is, using $[\![\_]\!]^{-}_{\mathscr{NC}}$. In addition, in [37], we have a lemma that is similar to Lemma 6.2 itself, but applies to states. Using that lemma, we can establish the equality above concerning $\underline{z}$ and $\underline{z'}$. Proceeding, with this equality and Proviso P1, a property of $\approx^{N}_{\mathscr{NC}}$ ensures that $\underline{(m, z)} \approx^{N}_{\mathscr{NC}} \underline{(m', z')}$. Then, by an argument similar to that of case (1), we can lift this result, given that $\underline{[\![z]\!]^{nops_1}_{\mathscr{N}}} = \underline{[\![z']\!]^{nops_2}_{\mathscr{N}}}$, to show $\underline{[\![m]\!]^{nops_1}_{\mathscr{SMD}}} = \underline{[\![m']\!]^{nops_2}_{\mathscr{SMD}}}$ as required. ■

Using the lemmas and results above, soundness can be established by a stepwise argument as follows, using Results R1 to R4 and Lemmas 6.1 and 6.2.

$$\underline{[\![m]\!]^{nops_1}_{\mathscr{SMD}}} \qquad \{\text{Definition of } \underline{m_0} \text{ and lemma 6.1}\}$$
$$= \underline{[\![m_0]\!]^{nops_1}_{\mathscr{SMD}}} \qquad \{\text{Lemma 6.2 assuming R1 to R4 hold}\}$$
$$= \underline{[\![m']\!]^{nops_2}_{\mathscr{SMD}}}$$

In summary, our overall strategy is to consider first an intermediate machine $\underline{m0}$, which includes the required interface, but no extra operation. Lemma 6.1 guarantees that the inclusion of the extra interface has no effect on the semantics. We note that although the new operation is in scope, it is not used, and so we do not need to consider its semantics. We consider the semantics of the new operation in the last step, justified by Lemma 6.2. This is our key result in this section.

6.2.2. *Law 6 (elim-deadline-transition)*

As proved in the previous section, the processes defined by $[\![\_]\!]^{-}_{\mathscr{SMD}}$ or $[\![\_]\!]^{-}_{\mathscr{OPD}}$ for the StateMachineDefs or OperationDefs $\underline{m}$ and $\underline{m'}$ identified in Law 2 have the same *tock*-CSP semantics. For Law 6, *tock*-CSP equality holds for the semantics of the StateMachineDefs characterized in the law. For OperationDefs, however, Law 6 establishes a weaker notion of equality, captured by a new relation $\approx^{\alpha}_{M}$ defined below, where α is a set of required variables. The key point in this case is that, as established below, if the semantics of two OperationDefs $\underline{m}$ and $\underline{m'}$ are related by $\approx^{\alpha}_{M}$, then any machines that differ just in that they call $\underline{m'}$ rather than $\underline{m}$ have the same semantics (either equal or related by $\approx^{\alpha}_{M}$). So, the weaker notion of equality is sufficient to justify transformations that replace $\underline{m}$ with $\underline{m'}$, whether they are StateMachineDefs or OperationDefs.

OperationDef relation We recall that within the semantics of machines (StateMachineDefs or OperationDefs), memory is modelled via CSP processes that offer to synchronize on channels to *get_* the current value of a variable or clock, or *set_* a new value. In Law 6, in the semantics for the machine on the left-hand side of the equality, the evaluation of the guard g, for example, occurs before the evaluation of the expression d for the deadline. For the machine on the right-hand side, g and d, used as arguments for dop˙ds, are evaluated in parallel. For StateMachineDefs, the *set_* and *get_* channels are all hidden in the semantics, and this kind of difference is not visible. For OperationDefs, however, whose semantics, we recall, is given by $[\![\_]\!]^{-}_{\mathscr{OPD}}$, both *get_* and *set_* events related to the required variables are exposed (see Fig. 49). So, in the case of Law 6, for example, if g and d involve required variables, the semantics of the OperationDefs on the left and right-hand sides are not equal.

In the semantics of any machine that calls such an operation, however, all *get_* events are hidden (see Fig. 49, where *set_* events show on the border of the outer box, but *get_* events do not). So, the semantics of a StateMachineDef that calls operations whose semantics are processes that can differ in the order of *get_* events, but nothing else, does not expose these differences. So, our notion of equality $=_S$ for OperationDefs does not require the order of *get_* events (for shared variables) to be the same. It is defined in terms of the new relation $\approx^{\alpha}_{M}$ over the semantics. To illustrate its role, we consider the following examples.

Example 9.

$$P_0 = get\_g?g \rightarrow get\_d?d \rightarrow get\_g?x \rightarrow e!\,(d{+}x) \rightarrow Skip$$
$$P_1 = get\_g?g \rightarrow get\_d?d \rightarrow e!\,(d{+}g) \rightarrow Skip$$

Initially, process P_0 is prepared to synchronize on *get_g* to receive a value *g*, followed by similar prefixings on *get_d* and *get_g* again. Afterwards, there is a prefixing on *e* with a value that is the sum of *d* and *x*, followed by termination. When we consider P_0's behaviour composed with a process modelling its memory, the only visible event is *e* with a value that depends only on the values communicated between P_0 and the memory, but not on the order of *get_*s. In that context, we can replace P_0 by P_1, where the second prefixing on *get_g* is removed, and the value of *g* read first is used in the output. In the context of a memory process, the behaviour is identical, so, we have that P_0 is related to P_1 by $\approx^{\{d,g\}}_{M}$. This, however, is only valid because there are no *set* or *share* events in between the *get_*s that could allow changes to values in the memory in between *share* events. We consider the following process P_2, similar to P_0, but including a *share* event.

Example 10.

$$P_2 = \begin{array}{l} get\_g?g \rightarrow get\_d?d \rightarrow share \rightarrow get\_g?x \rightarrow \\ e!\,(d{+}x) \rightarrow Skip \end{array}$$

In this example, in between the two *get_g* events, we have a synchronization on *share*. With this, it means that the value of *g* could change between the *get_*s, and we cannot disregard the second *get_g*. The same holds, if, instead of *share*, we had a *set_g* event.

The relation $P \approx^{\alpha}_{M} Q$ is defined below; it compares processes $LMem(P, \underline{\alpha})$ and $LMem(Q, \underline{\alpha})$ instead of P and Q directly. With $LMem(P, \underline{\alpha})$ (and similarly, $LMem(Q, \underline{\alpha})$) what we have is P with a copy of its associated memory for required variables, and with the internal communications with that memory hidden. So, the comparison between P and Q defined by $P \approx^{\alpha}_{M} Q$ is restricted to its behaviour in context.

Figure 51 shows the process $LMem(P, \underline{\alpha})$, where we use αv to denote a particular required variable. The memory process *LMemV* defines a context for P and has the following roles. (1) It is always ready to interact with the environment on a *get_αv* channel, whenever the value of v is known (set, as determined by *αvset*). (2) It holds a local value of αv, which is used by P, via *getL_αv* events, instead of *get_αv* events directly. (3) It, therefore, hides any order

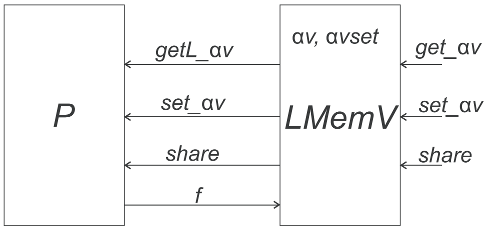


Figure 51. Definition 6.3 – Pictorial representation of $LMem(P, \{\alpha v\})$: the process $LMemV$ records the value of αv and whether its value is initialized using a boolean $\alpha vset$.

on $get\_\alpha v$ events imposed by P. (4) It passes on any *share* events, recording that the value of αv is then not known ($\alpha vset$ is false) as it may have been changed by other processes. (5) It passes on $set\_\alpha v$ events, recording that the value of αv is now set. An extra channel f is used by P to terminate the memory process, if P itself finishes.

For example, if we assume that g and d are required variables, we have the following results. $LMem(P_0, \{g, d\})$ and $LMem(P_1, \{g, d\})$ are the same, because after the initial *get_*s neither process offers to synchronize in *set_*s or *share*, and so the following (internal) *getL_*s happen without further interference. $LMem(P_2, \{g, d\})$, on the other hand, behaves differently. Although its initial behaviour is the same as that of $LMem(P_0, \{d, g\})$ and $LMem(P_1, \{d, g\})$, after synchronizing on *share* the memory process must refresh its copy of the values of d and g before the next $getL\_g$, namely via additional synchronizations on get_d and get_g.

The definition below formalizes $P \approx^{\alpha}_{M} Q$ in terms of $LMem$ as suggested by the examples. Formally, in $LMem(P, \underline{\alpha})$ we have a parallel composition of P, where, for each variable $\underline{v}$ in $\underline{\alpha}$, $get\_\underline{id(v)}$ is renamed to $getL\_\underline{id(v)}$, with a process $\underline{\mathbf{LMemV}(\alpha)}$, that models the local copy of the state of variables in $\underline{\alpha}$ by synchronizing with P on events *getL_* instead of *get_* (see Fig. 51). P is sequentially composed with a prefixing on f, a fresh event, to terminate the parallel composition with $\underline{\mathbf{LMemV}(\alpha)}$ after P terminates. The synchronization set includes events $getL\_\underline{id(v)}$ and $set\_\underline{id(v)}$ for all variables $\underline{v}$ in $\underline{\alpha}$, f, and *share*. The hiding of events *getL_* and f completes the definition. $\underline{\mathbf{LMemV}(\alpha)}$ is defined in [37].

Definition 6.3.

$$P \approx^{\alpha}_{M} Q \iff LMem(P, \underline{\alpha}) = LMem(Q, \underline{\alpha})$$

where

$$LMem(P, \underline{\alpha}) = \left(\begin{array}{l} (P\,[\![\, \underline{v} : \alpha \bullet get\_\underline{id(v)} \leftarrow getL\_\underline{id(v)} \,]\!]\ ; f \rightarrow Skip) \\ [\![\{\!|\underline{v} : \alpha \bullet getL\_\underline{id(v)}, set\_\underline{id(v)}|\!\} \cup \{f, share\}]\!] \\ \underline{\mathbf{LMemV}(\alpha)} \end{array}\right)$$

$$\setminus \{\!|\underline{v} : \alpha \bullet getL\_\underline{id(v)}|\!\} \cup \{f\}$$

Using $\approx^{\alpha}_{M}$ we can then define another relation $=_S$ for comparing OperationDefs that differ in their *get_*s.

Definition 6.4.

$$op_1 =_S^{(\alpha, nops_1, nops_2)} op_2 \iff [\![op_1]\!]^{nops_1}_{\mathscr{OPD}} \approx^{\alpha}_{M} [\![op_2]\!]^{nops_2}_{\mathscr{OPD}}$$

Operations op_1 and op_2 are related by $=_S^{(\alpha, nops_1, nops_2)}$ exactly when their semantics, given by $[\![\_]\!]^{-}_{\mathscr{OPD}}$, is related by $\approx^{\alpha}_{M}$. Here, $\underline{\alpha}$ is a set of variables and $nops_1$ and $nops_2$ are functions that define the semantics of operations that op_1 and op_2 may call. (We consider in the above definition just the simpler case in which op_1 and op_2 do not have parameters, so that $[\![op_1]\!]^{nops_1}_{\mathscr{OPD}}$ and $[\![op_2]\!]^{nops_2}_{\mathscr{OPD}}$ are processes, rather than functions from arguments to processes. The generalization to require $\approx^{\alpha}_{M}$ to hold for each combination of arguments is simple.) Importantly, Theorem 6.1 presented below establishes that the semantics of StateMachineDefs is unaffected by exchanging the definitions of an operation op if they are related by $=_S^{(\alpha, nops_1, nops_2)}$.

Theorem 6.1. Provided $\underline{op_1} =_S^{(\alpha, nops, nops)} \underline{op_2}$, where $\underline{op_1}$ and $\underline{op_2}$ are OperationDefinitions with the same name op, $\underline{\alpha}$ is the set of variables required by $\underline{op_1}$ and $\underline{op_2}$, and *nops* defines operations other than op, then for any StateMachineDef $\underline{m}$,

$$[\![\underline{m}]\!]^{nops \cup \{op \mapsto [\![\underline{op_1}]\!]^{nops}_{\mathscr{OPD}}\}}_{\mathscr{SMD}} = [\![\underline{m}]\!]^{nops \cup \{op \mapsto [\![\underline{op_2}]\!]^{nops}_{\mathscr{OPD}}\}}_{\mathscr{SMD}}$$

Proof. Let $\underline{\mathbf{nops_1}} = nops \cup \{op \mapsto [\![\underline{op_1}]\!]^{nops}_{\mathscr{OPD}}\}$ and $\underline{\mathbf{nops_2}} = nops \cup \{op \mapsto [\![\underline{op_2}]\!]^{nops}_{\mathscr{OPD}}\}$. There are two cases:

1) op is not called by $\underline{m}$: so the semantics of $\underline{op_1}$ and $\underline{op_2}$ do not affect that of the state machine $\underline{m}$, and so $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMD}} = [\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{SMD}}$, as required.
2) op is called by $\underline{m}$: because $\underline{m}$ is well-formed, all variables required by $\underline{op_1}$ and $\underline{op_2}$ are either defined or required by $\underline{m}$, that is, $\underline{\alpha}$ is a subset of $\underline{\mathbf{localVariables(m)}} \cup \underline{\mathbf{requiredVariables(m)}}$. So, let $\underline{\alpha} = \underline{\beta} \cup \underline{\gamma}$, where $\underline{\beta}$ and $\underline{\gamma}$ are disjoint sets, $\underline{\beta}$ is a subset of local variables of $\underline{m}$ and $\underline{\gamma}$ is a subset of required variables of $\underline{m}$. Therefore, from the proviso, we have that $\underline{op_1} =_S^{(\beta \cup \gamma, nops, nops)} \underline{op_2}$, and by definition $[\![\underline{op_1}]\!]^{\underline{\mathbf{nops}}}_{\mathscr{OPD}} \approx^{\beta \cup \gamma}_{M} [\![\underline{op_2}]\!]^{\underline{\mathbf{nops}}}_{\mathscr{OPD}}$. Therefore, the semantics of calls to $\underline{op_1}$ and $\underline{op_2}$, given by the copy rule, is likewise related by $\approx^{\beta \cup \gamma}_{M}$. The result then follows by three lemmas that lift this result (see Fig. 49) to (1) the NodeContainer semantics of $\underline{m}$ (to obtain $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{NC}} \approx^{\beta \cup \gamma}_{M} [\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{NC}}$), (2) from this to the StateMachineBody semantics of $\underline{m}$, (to obtain $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}} \approx^{\gamma}_{M} [\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{SMB}}$), and finally (3) to the StateMachineDef semantics of $\underline{m}$, obtaining $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMD}} = [\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{SMD}}$ as required. ■

The lemmas used in the above proof are in [37], and we present below the last of these for illustration.

Lemma 6.3. Provided $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}} \approx^{\alpha}_{M} [\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{SMB}}$, where $\alpha \subseteq \underline{\mathbf{requiredVariables(m)}}$, then $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMD}} = [\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{SMD}}$.

Proof. We recall that the StateMachineDef semantics for $\underline{m}$ is defined by a parallel composition of its StateMachineBody semantics and the memory process for required variables, with the *get_* events hidden. In that composition (by a lemma in [37]) we can replace $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}}$ with $LMem([\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}}, \underline{\alpha})$ because $LMem([\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}}, \underline{\alpha})$ only places, between $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}}$ and the memory process, an $LMemV$ process (see Fig. 51) that caches a local value for the required variables and passes every other communication on to $[\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}}$. From the proviso and the definition of $\approx_M$ we have that $LMem([\![\underline{m}]\!]^{\underline{\mathbf{nops_1}}}_{\mathscr{SMB}}, \underline{\alpha}) = LMem([\![\underline{m}]\!]^{\underline{\mathbf{nops_2}}}_{\mathscr{SMB}}, \underline{\alpha})$. So, we can make the following argument, where we indicate the parallel composition using $\parallel$, the memory process as MP, and the set of channels being hidden as $\{\!|get\_, \ldots|\!\}$.

$$
\begin{aligned}
&\underline{[\![m]\!]^{nops_1}_{\mathscr{SMD}}}\\
&= (\underline{[\![m]\!]^{nops_1}_{\mathscr{SMB}}} \parallel MP) \setminus \{\!|get\_, \ldots|\!\}\\
&= (LMem(\underline{[\![m]\!]^{nops_1}_{\mathscr{SMB}}}, \underline{\alpha}) \parallel MP) \setminus \{\!|get\_, \ldots|\!\}\\
&= (LMem(\underline{[\![m]\!]^{nops_2}_{\mathscr{SMB}}}, \underline{\alpha}) \parallel MP) \setminus \{\!|get\_, \ldots|\!\}\\
&= (\underline{[\![m]\!]^{nops_2}_{\mathscr{SMB}}} \parallel MP) \setminus \{\!|get\_, \ldots|\!\}\\
&= \underline{[\![m]\!]^{nops_2}_{\mathscr{SMD}}}
\end{aligned}
$$
■

With Theorem 6.1, we ensure that application of Law 6 to operations called in StateMachineDefs is sound, because, as shown below, Law 6 establishes $=_S$.

Law 6, stated diagrammatically in Section 4.1, is formalized and proved sound below. As mentioned, the equality holds for StateMachineDefs, while for OperationDefs we have the relation $\approx^{\underline{rv}}_{M}$, where $\underline{rv}$ is the subset of variables required by $\underline{m}$ and used in the guard or deadline of a transition whose deadline is removed by application of Law 6 (from left to right).

Formalisation 2 (Law 6).

$$\underline{[\![m\colon \mathrm{StateMachineDef}]\!]^{nops_1}_{\mathscr{SMD}}} = \underline{[\![m'\colon \mathrm{StateMachineDef}]\!]^{nops_2}_{\mathscr{SMD}}}$$

where

- $\underline{S}, \underline{S'}$: State, $\underline{ds}$: Statement, $\underline{Idop\_ds}$: Interface,
- $\underline{action_{ds}}$: During, $\underline{td}$: Transition,
- $\underline{nc}$: NodeContainer,
- $\underline{dop\_ds}$: OperationDef, $\underline{call_{dop\_ds}}$: Call

A1. $\underline{action_{ds} \in S.actions \land ds = action_{ds}.action}$
A2. $\underline{S \in nc.nodes}$
A3. $\left(\begin{array}{l} td \in nc.transitions \land td.deadline \neq null \land \\ td.source = S \end{array}\right)$
A4. $\underline{dop\_ds = dop\_dsOperation(ds, td)}$
A5. $\underline{Idop\_ds = \{operations \mapsto \{opSig(dop\_ds)\}\}}$
A6. $\left(\begin{array}{l} call_{dop\_ds}.op = opSig(dop\_ds) \land \\ call_{dop\_ds}.args = usedArgs(d, td.condition, ds) \end{array}\right)$
A7. $\underline{nops_2 = nops_1 \cup \{dop\_ds.name \mapsto [\![dop\_ds]\!]_{\mathscr{OPD}}\}}$
A8. $S' = S \oplus \left\{\begin{array}{l} actions \mapsto \\ \quad (S.actions \setminus \{action_{ds}\}) \\ \quad \cup \{action_{ds} \oplus \{action \mapsto call_{dop\_ds}\}\} \end{array}\right\}$
A9. $\underline{(m \oplus \{RInterfaces \mapsto m.RInterfaces \cup Idop\_ds\}, nc)}$
$\approx^{N}_{\mathscr{NC}}$
$\left(m', nc \oplus \left\{\begin{array}{l} nodes \mapsto (nc.nodes \setminus \{S\}) \cup \{S'\} \\ trans \mapsto \\ \quad (nc.trans \setminus \{td\}) \cup \\ \quad \{\phi(S, S')(td \oplus \{deadline \mapsto null\})\} \end{array}\right\}\right)$ ■

Similarly to the formalization of Law 2, we have the declaration of metavariables used in Law 6. Namely, we declare $\underline{S}$, $\underline{ds}$, $\underline{Idop\_ds}$, $\underline{action_{ds}}$ and $\underline{td}$, which capture model elements of the state machines. We use $\underline{S'}$ to refer to the state on the right-hand side of Law 6. The NodeContainer $\underline{nc}$ is used to capture the State, StateMachineDef or OperationDef that contains the state $\underline{S}$ and its outgoing transition $\underline{td}$. As before, in the diagram for Law 6 we use $\underline{dop\_s(\ldots)}$ to refer both to a call to an operation and to the signature used in its definition. Here we declare $\underline{call_{dop\_s}}$ to represent the call, and $\underline{dop\_s}$ to represent the operation itself.

Following the declarations, there is a list of numbered properties restricting the values of the metavariables. First, in A1, we record the fact that Law 6 uses $\underline{action_{ds}}$ as a during action of $\underline{S}$ and that $\underline{ds}$ is the statement of $\underline{action_{ds}}$. The case where $\underline{S}$ has no during action is omitted for simplicity, but can be accounted, for example, by considering $\underline{ds}$ as skip, which terminates immediately. A2 requires $\underline{S}$ to be one of $\underline{nc}$'s states. A3 states that $\underline{td}$ is one of $\underline{nc}$'s transitions with a deadline and whose source is State $\underline{S}$. A4 formalizes the definition of $\underline{dop\_ds}$ using a metafunction $\underline{dop\_dsOperation(ds, td)}$, defined in [37]; it defines an operation for the given statement ds and transition td, using interfaces req_d_g_ds, defEvents_ds and Idop_ds and IdeadlineCheck. A5 formalizes the definition of the interface Idop_ds, where its component $\underline{operations}$ maps to the singleton set containing the signature of dop_ds. A6 records the Call's target as the operation $\underline{call_{dop\_s}}$ and that the arguments to be passed, if any, are given by a function $\underline{usedArgs(\ldots)}$. A7 states that the parameter $\underline{nops_2}$ is defined by extending $\underline{nops_1}$ with the semantics of dop_ds. A8 captures how $\underline{S'}$ is defined in terms of $\underline{S}$, where the original action $\underline{action_{ds}}$ is replaced by $\underline{call_{dop\_s}}$. Finally, A9 captures how the machine $\underline{m'}$ is defined in terms of $\underline{m}$, after it is enriched with the required interface $\underline{Idop\_s}$. The difference in $\underline{m'}$ is that the NodeContainer $\underline{nc}$, a descendant state of $\underline{m}$ or itself $\underline{m}$, has $\underline{S}$ removed and a new state $\underline{S'}$ added; and $\underline{td}$ removed and replaced by a transition where the $\underline{deadline}$ is set to $\underline{null}$ and any use of $\underline{S}$ is replaced by $\underline{S'}$ using ϕ, defined previously in Section 6.2.1. The proof of soundness for Law 6 in [37] is sketched below.

Proof. There are two cases to consider: $\underline{m}$ and $\underline{m'}$ are StateMachineDefs or OperationDefs. We focus on the latter given that equality for StateMachineDefs can be established by a further application of Lemma 6.3. For

- $\underline{m_0 = m \oplus \{RInterfaces \mapsto m.RInterfaces \cup Idop\_ds\}}$,
- $\underline{td_0 = \phi(S, S')(td \oplus \{deadline \mapsto null\})}$,
- $\underline{a_0 = \{action_{ds} \oplus \{action \mapsto call_{dop\_ds}\}\}}$,
- $\underline{nc_0} = nc \oplus \left\{\begin{array}{l} nodes \mapsto (nc.nodes \setminus \{S\}) \cup \{S'\}, \\ trans \mapsto (nc.trans \setminus \{td\}) \cup \{td_0\} \end{array}\right\}$
- $\underline{uv} = \left(\begin{array}{l} usedVariables(td.condition) \\ \cup usedVariables(td.deadline) \end{array}\right)$

we can establish the following results.

R1. $(\underline{m_0}, \underline{nc}) \approx^{N}_{\mathscr{NC}} (\underline{m'}, \underline{nc_0})$;
R2. $\underline{vars(nc)} = \underline{vars(nc_0)}$;
R3. $\underline{usedOps(S')} = \underline{usedOps(S) \cup \{dop\_ds\}}$, that is, the set of operations used by $\underline{S'}$, those called by actions of $\underline{S'}$, its substates or transitions, is that used by $\underline{S}$ augmented with $\underline{dop\_ds}$, thus we also have that $\underline{usedOps(m')} = \underline{usedOps(m_0) \cup \{dop\_ds\}}$, and $\underline{usedOps(m') \cap usedOps(m_0)} = \underline{usedOps(m_0)}$, that is, the operations used by both $\underline{m'}$ and $\underline{m_0}$ are the same. Therefore, they behave the same:

$$\forall \underline{op} : \underline{usedOps(m')} \cap \underline{usedOps(m_0)} \bullet$$
$$\underline{[\![nops_1(op)]\!]^{nops_1}_{\mathscr{OPD}}} = \underline{[\![nops_2(op)]\!]^{nops_2}_{\mathscr{OPD}}}$$

R3. $\underline{[\![nc]\!]^{nops_1}_{\mathscr{NC}}} \approx^{uv}_{M} \underline{[\![nc_0]\!]^{nops_2}_{\mathscr{NC}}}$, that is, the semantics of the NodeContainers $\underline{nc_0}$ and $\underline{nc}$ is related by $\approx^{\underline{uv}}_{M}$, where $\underline{uv}$ are variables used in the condition or deadline of $\underline{td}$, and $\underline{rv} \subseteq \underline{requiredVars(m) \cap uv}$, the subset of $\underline{uv}$ that is required by

$\underline{\mathrm{m}}$. Proving this result requires showing that the parallel composition of the semantics of $\underline{\mathrm{S}}$ with its transition ($\underline{\mathrm{td}}$) deadline semantics is equivalent to that of $\underline{\mathrm{S}'}$, where there is no deadline on transition $\underline{\mathrm{td}_0}$ and instead the during action is replaced by a call to dop˙ds.

They justify the application of the following key lemma.

Lemma 6.4

$$\underline{\llbracket \mathtt{m}:\mathtt{NodeContainer} \rrbracket^{\mathrm{nops}_1}_{\mathscr{NC}}} \approx^{\alpha}_{\mathrm{M}} \underline{\llbracket \mathtt{m}':\mathtt{NodeContainer} \rrbracket^{\mathrm{nops}_2}_{\mathscr{NC}}}$$

provided:

P1. $\underline{(\mathrm{m}, \mathrm{nc}) \approx^{\mathrm{N}}_{\mathscr{NC}} (\mathrm{m}', \mathrm{nc}')}$
P2. $\underline{\mathrm{vars}(\mathrm{nc}) = \mathrm{vars}(\mathrm{nc}')}$
P3. $\left(\begin{array}{l} \forall \underline{\mathrm{op}} : \underline{\mathrm{usedOps}(\mathrm{m}')} \cap \underline{\mathrm{usedOps}(\mathrm{m})} \bullet \\ \underline{\llbracket \mathrm{nops}_1(\mathrm{op}) \rrbracket^{\mathrm{nops}_1}_{\mathscr{OPD}}} = \underline{\llbracket \mathrm{nops}_2(\mathrm{op}) \rrbracket^{\mathrm{nops}_2}_{\mathscr{OPD}}} \end{array} \right)$
P4. $\underline{\llbracket \mathrm{nc} \rrbracket^{\mathrm{nops}_1}_{\mathscr{NC}}} \approx^{\alpha}_{\mathrm{M}} \underline{\llbracket \mathrm{nc}' \rrbracket^{\mathrm{nops}_2}_{\mathscr{NC}}}$

Proof. The proof strategy is similar to that in the proof of Lemma 6.2. The base case, when $\underline{\mathrm{m} = \mathrm{nc}}$ and $\underline{\mathrm{m}' = \mathrm{nc}'}$ hold directly, with the other cases established by induction using the definitions of $\underline{\llbracket \_ \rrbracket_{\mathscr{NC}}}$ and $\underline{\approx^{\mathrm{N}}_{\mathscr{NC}}}$, and the semantics of composite states. ■

Using the lemmas and results above, soundness can be established by a stepwise argument as follows, using Results R1 to R4, a version of Lemma 6.1 for OperationDefs, and Lemma 6.4. First, we have that $\underline{\llbracket \mathrm{m} \rrbracket^{\mathrm{nops}_1}_{\mathscr{SMB}}}$ is equal to $\underline{\llbracket \mathrm{m}_0 \rrbracket^{\mathrm{nops}_1}_{\mathscr{SMB}}}$ using the definition of $\underline{\mathrm{m}_0}$ and Lemma 6.1. So the proof requires showing that $\underline{\llbracket \mathrm{m}_0 \rrbracket^{\mathrm{nops}_1}_{\mathscr{SMB}}}$ and $\underline{\llbracket \mathrm{m}' \rrbracket^{\mathrm{nops}_2}_{\mathscr{SMB}}}$ are related by $\approx_{\mathrm{M}}$. By a Lemma in [37], we have $\underline{\llbracket \mathrm{nc} \rrbracket^{\mathrm{nops}_1}_{\mathscr{NC}}} \approx^{\underline{\mathrm{uv}}}_{\mathrm{M}} \underline{\llbracket \mathrm{nc}_0 \rrbracket^{\mathrm{nops}_2}_{\mathscr{NC}}}$, and so using Lemma 6.4 we have $\underline{\llbracket \mathrm{m}_0 \rrbracket^{\mathrm{nops}_1}_{\mathscr{NC}}} \approx^{\underline{\mathrm{uv}}}_{\mathrm{M}} \underline{\llbracket \mathrm{m}' \rrbracket^{\mathrm{nops}_2}_{\mathscr{NC}}}$. Finally, this result can be lifted to show that $\underline{\llbracket \mathrm{m}_0 \rrbracket^{\mathrm{nops}_1}_{\mathscr{SMB}}} \approx^{\underline{\mathrm{rv}}}_{\mathrm{M}} \underline{\llbracket \mathrm{m}' \rrbracket^{\mathrm{nops}_2}_{\mathscr{SMB}}}$, as required, using a lifting from the NodeContainer semantics to the StateMachineBody semantics as used in case 2 of Theorem 6.1's proof.

Finally, we note that if $\underline{\mathrm{m}}$ and $\underline{\mathrm{m}'}$ are StateMachineDefs, we can further apply Lemma 6.3 to obtain equality.

7. CONCLUSIONS

This paper presents what is, as far as we know, the first set of transformations for timed UML-like state machines, with accompanying proofs of soundness and notions of completeness based on normal forms. We consider machines in both a closed (RoboChart machines) and an open context. Most importantly, we consider machines that can specify time properties. Moreover, our laws and normalization procedures are implemented to provide practical normalization engines.

RoboChart is unique as a statechart-based notation with support for time modelling and closed components, and a formal semantics. The open state machines are closer to those in traditional Statecharts [49], UML [5] and SysML [50], but also support a rigorous approach to time modelling. This unique approach, with a process algebraic semantics, can be incorporated in other statechart-based notations. In this sense, our results are concerned with a time modelling approach, rather than just RoboChart or even open machines.

On the other hand, our notations do not include the whole repertoire of constructs of UML and SysML, for example. To keep formalization of models tractable, we have eliminated facilities such as inter-level transitions, which make compositional reasoning difficult, if not impossible. Writing models that require elaborate control flows that are not directly supported is normally possible via an encoding using variables. For example, we can model completion events using boolean variables to guard transitions and ensure they are enabled only when all internal activity of a state has finished.

Our laws embed a reasoning and normalization approach that can inform similar techniques for other notations. Those that offer more restricted facilities to deal with data and states, such as automata, can benefit from simpler versions of the laws. (Timed automata [51] provide extensive support for time modelling, but not for structured modelling, using rich data, state hierarchies and action-based control.) Reliance on our soundness argument requires a process-algebraic semantics, but considering soundness in other semantic contexts is an interesting avenue of further study.

To prove soundness of our laws, it has been beneficial to rework the semantics of RoboChart state machines to provide a compositional formulation. Two aspects have been changed. The semantics of operations is now captured by the copy rule, and the semantics of a composite state is given just in terms of that of its state machine. As a result, the notion of NodeContainer is given a semantics in its own right, and this provides a direct connection with the semantics of open machines. All this simplifies the semantics, facilitating proof, and, as it turns out, reducing the number of states of the CSP model and improving efficiency of model checking.

Previous work on refinement laws for state machines has been carried out in the context of SysML [52]. In that approach, a notion of refinement is defined for state machines in the context of block diagrams, which are used to define systems (and their components) in SysML. Similarly, here we define refinement for machines in the context of modules. The objective in [52] is to compare systems at different levels of abstraction. Here, we restrict ourselves to equalities, but do define equality as mutual refinement. To enable refinement reasoning, extensions to SysML are proposed in [52] to support, for example, hiding and a well-defined action language based on CSP. Hiding is already available in RoboChart (via block containment), since RoboChart is a language developed to support formal verification (by refinement). Consequently, some of the laws in [52] potentially have a counterpart as laws of open machines, and vice versa. We note, however, that the SysML machines do not have time constructs.

In [53], the authors propose a set of laws for UML-RT [54], a UML profile with a clear definition for reactive components and component protocols, useful to describe concurrent and distributed domains. The focus of the laws is on the new elements that UML-RT adds to UML: active classes (capsules), protocols, ports and connections. Laws are not concerned with UML-RT statecharts in their own right, but in the context of the decomposition of active classes. The normal form removes parallelism. That work can be considered complementary to our contribution here, as it addresses concurrency, but not time constructs.

To summarize, our work considers a rich state-machine notation in terms of its support to specify time properties. As far as we know, we provide the only set of sound and complete laws for timed state machines.

DATA AVAILABILITY STATEMENT

Supporting material is available in [37] yellowand robostar.cs.york.ac.uk as explicitly indicated in the body of the paper.

ACKNOWLEDGMENTS

Our work is funded by Brazilian Research Councils, via project INES, grants CNPq/465614/2014-0 and FACEPE/APQ/0388-1.03/14, the Royal Academy of Engineering, grant CiET1718/45 and the UK EPSRC, grants EP/R025479/1 and EP/V026801/1. We are also grateful to members of the RoboStar (robostar.cs.york.ac.uk) centre and to anonymous referees for useful discussions and suggestions.

REFERENCES

1. Park, H.W., Ramezani, A. and Grizzle, J.W. (2013) A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. *IEEE Trans. Robot.*, **29**, 331–345.
2. Rabbath, C.A. (2013) A finite-state machine for collaborative airlift with a formation of unmanned air vehicles. *J. Intell. Robot. Syst.*, **70**, 233–253.
3. Tomic, T., Schmid, K., Lutz, P., Domel, A., Kassecker, M., Mair, E., Grixa, I.L., Ruess, F., Suppa, M. and Burschka, D. (2012) Toward a fully autonomous UAV: research platform for indoor and outdoor urban search and rescue. *IEEE Robot. Autom. Mag.*, **19**, 46–56.
4. The MathWorks, Inc Stateflow and Stateflow coder 7 User's guide. www.mathworks.com/products.
5. OMG (2015) OMG *Unified Modeling Language*.
6. OMG (2012) OMG systems Modeling language (OMG SysML). *Version*, **1**, 3.
7. Hoare, C.A.R., Hayes, I.J., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M. and Sufrin, B.A. (1987) Laws of programming. *Commun. ACM*, **30**, 672–686.
8. Morgan, C.C. (1994) *Programming from Specifications* (2nd edn). Prentice-Hall.
9. Jifeng, H. and Bowen, J. (1994) Specification, verification, and prototyping of an optimized compiler. *Form. Asp. Comput.*, **6**, 643–658.
10. Sampaio, A.C.A. (1997) *An Algebraic Approach to Compiler Design, AMAST Series in Computing,* **4**. World Scientific.
11. Duran, A., Cavalcanti, A.L.C. and Sampaio, A.C.A. (2010) An algebraic approach to the Design of Compilers for object-oriented languages. *Form. Aspects Comput.*, **22**, 489–535.
12. Fowler, M. (1999) *Refactoring*. Addison-Wesley.
13. Opdyke, W. (1992) *Refactoring Object-oriented Frameworks* PhD thesis. University of Illinois at Urban Champaign.
14. Cornélio, M., Cavalcanti, A.L.C. and Sampaio, A.C.A. (2010) Sound Refactorings. *Sci. Comput. Program.*, **75**, 106–133.
15. Roscoe, A.W. and Hoare, C.A. (1988) The Laws of occam programming. *Theor. Comput. Sci.*, **60**, 177–229.
16. Bird, R. and de Moor, O. (1997) *Algebra of Programming*. Prentice-Hall.
17. Seres, S., Spivey, M. and Hoare, T. (1999) Algebra of logic programming. *ICPL'99*..
18. Borba, P.H.M., Sampaio, A.C.A., Cavalcanti, A.L.C. and Cornélio, M.L. (2004) Algebraic reasoning for object-oriented programming. *Sci. Comput. Program.*, **52**, 53–100.
19. Zeyda, F. and Cavalcanti, A.L.C. (2015) Laws of mission-based programming. *Form. Asp. Comput.*, **27**, 423–472.
20. Perna, J.I., Woodcock, J.C.P., Sampaio, A.C.A. and Iyoda, J. (2011) Correct hardware synthesis - an algebraic approach. *Acta Inform.*, **48**, 363–396.
21. Lano, K. and Evans, A. (1999) Rigorous development in uml. In Finance, J.-P. (ed) *Fundamental Approaches to Software Engineering, Berlin, Heidelberg*, pp. 129–143. Springer, Berlin Heidelberg.
22. Breu, R., Grosu, R., Huber, E., Rumpe, B. and Schwerin, W. (1998) Systems, views and models of uml. In Schader, M., Korthaus, A. (eds) *The Unified Modeling Language*, pp. 93–108. Physica-Verlag HD.
23. Broy, M., Cengarle, M.V. and Rumpe, B. (2007) *Semantics of UML - Towards a System Model for UML: The State Machine Model* Technical Report TUM-I0711. Institut für Informatik, Technische Universität München.
24. Kuske, S., Gogolla, M., Kollmann, R. and Kreowski, H.-J. (2002) An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In Butler, M., Petre, L., SereKaisa, K. (eds) *Integrated Formal Methods, Lecture Notes in Computer Science* (Vol. **2335**), pp. 11–28. Springer.
25. Café, D.C., dos Santos, F.V., Hardebolle, C., Jacquet, C. and Boulanger, F. (2013) Multi-paradigm semantics for simulating sysml models using systemc-ams. *Forum Specification Des. Lang.*, 1–8.
26. Davies, J. and Crichton, C. (2003) Concurrency and refinement in the unified modeling language. *Form. Asp. Comput.*, **15**, 118–145.
27. Rasch, H. and Wehrheim, H. (2003) Checking consistency in UML diagrams: Classes and state machines. In Najm, E., Nestmann, U., Stevens, P. (eds) *Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science* (Vol. **2884**), pp. 229–243. Springer.
28. Lima, L., Miyazawa, A., Cavalcanti, A.L.C., Cornélio, M., Iyoda, J., Sampaio, A.C.A., Hains, R., Larkham, A. and Lewis, V. (2017) An integrated semantics for reasoning about SysML design models using refinement. *Softw. Syst. Model.*, **16**, 875–902.
29. Miyazawa, A. and Cavalcanti, A.L.C. (2012) Refinement-oriented models of Stateflow charts. *Sci. Compu. Program.*, **77**, 1151–1177.
30. Bergstra, J.A. and Ponse, A. (2002) Combining programs and state machines. *J. Log. Algebr. Program.*, **51**, 175–192.
31. Brunner, S. G., Steinmetz, F., Belder, R., and Domel, A. (2016) Rafcon: a graphical tool for engineering complex, robotic tasks. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3283–3290.
32. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J. and Woodcock, J.C.P. (2019) RoboChart: modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.*, **18**, 3097–3149.
33. Nordmann, A., Hochgeschwender, N., Wigand, D. and Wrede, S. (2016) A survey on domain-specific modeling and languages in robotics. *J. Softw. Eng. Robot.*, **7**, 75–99.
34. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A. L. C., and Timmis, J. (2017) Automatic property checking of robotic applications. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3869–3876.
35. Roscoe, A.W. (1998) *The Theory and Practice of Concurrency Prentice-Hall Series in Computer Science*. Prentice-Hall.
36. Davidson, J. R., Silwal, A., Hohimer, C. J., Karkee, M., Mo, C., and Zhang, Q. (2016) Proof-of-concept of a robotic apple harvester. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 634–639.
37. Cavalcanti, A.L.C., Filho, M.C., Ribeiro, P. and Sampaio, A.C.A. (2022) *Laws of timed state machines – extended version* Technical report. RoboStar Centre on software engineering for robotics Available at robostar.cs.york.ac.uk/publications/techreports/reports/CCFRS22.pdf.
38. Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N., and Polack, F. A. C. (2009) The design of a conceptual framework and technical infrastructure for model management language engineering.

2009 *14th IEEE international conference on engineering of complex computer systems*, pp. 162–171.
39. Ye, K., Cavalcanti, A.L.C., Foster, S., Miyazawa, A. and Woodcock, J.C.P. (2021) Probabilistic modelling and verification using RoboChart and PRISM. *Softw. Syst. Model.*, **21**, 667–716.
40. Gibson-Robinson, T., Armstrong, P., Boulgakov, A. and Roscoe, A.W. (2014) FDR3 - a modern refinement checker for CSP. *Tools Algorithms Constr. Anal. Syst.*, 187–201.
41. Kwiatkowska, M., Norman, G. and Parker, D. (2004) Probabilistic symbolic model checking with PRISM: a hybrid approach. *Int. J. Softw. Tools Technol. Transfer*, **6**, 128–142.
42. Dixon, C., Winfield, A.F.T., Fisher, M. and Zeng, C. (2012) Towards temporal verification of swarm robotic systems. *Robot. Auton. Syst.*, **60**, 1429–1441.
43. Cavalcanti, A.L.C., Sampaio, A.C.A., Miyazawa, A., Ribeiro, P., Filho, M.C., Didier, A., Li, W. and Timmis, J. (2019) Verified simulation for robotics. *Sci. Comput. Program.*, **174**, 1–37.
44. Baxter, J., Ribeiro, P. and Cavalcanti, A.L.C. (2022) Sound reasoning in tock-CSP. *Acta Inform.*, **59**, 125–162.
45. Milner, A.J.R.G. (1983) Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, **25**, 267–310.
46. Milner, R. (1999) *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press.
47. Bergstra, J.A. and Klop, J.W. (1985) Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, **37**, 77–121.
48. Miyazawa, A., Ribeiro, P., Ye, K., Cavalcanti, A.L.C., Li, W., Timmis, J. and Woodcock, J.C.P. (2020) *RoboChart: Modelling, Verification and Simulation for Robotics* Technical report. University of York, Department of Computer Science, York, UK Available at www.cs.york.ac.uk/robostar/notations/.
49. Harel, D. (1987) Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, **8**, 231–274.
50. OMG (2017). OMG systems Modeling language (OMG SysML), *Version* **2**.0.
51. Alur, R. and Dill, D.L. (1994) A theory of timed automata. *Theor. Comput. Sci.*, **126**, 183–235.
52. Miyazawa, A. and Cavalcanti, A.L.C. (2014) Refinement-based verification of implementations of Stateflow charts. *Form. Asp. Comput.*, **26**, 367–405.
53. Ramos, R., Sampaio, A. C. A., and Mota, A. C. (2006) Transformation laws for UML-RT. In Gorrieri, R. and Wehrheim, H. (eds.), *8th IFIP WG 6.1 International Conference onFormal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science*, **4037**, pp. 123–137. Springer.
54. Selic, B. and Rumbaugh, J. (1998) *Using UML for modeling complex real-time systems* Technical report. ObjecTime Limited.