Empowering SysML-Based Software Architecture Description with Formal Verification: From SysADL to CSP

Fagner Dias[1], Marcel Oliveira[1(✉)], Thais Batista[1], Everton Cavalcante[1], Jair Leite[1], Flavio Oquendo[2], and Camila Araújo[1,3]

[1] DIMAp, Federal University of Rio Grande do Norte, Natal, Brazil
fagnerdiasmorais@gmail.com, {marcel,thais,everton,jair}@dimap.ufrn.br, cmlaraujo@gmail.com

[2] IRISA-UMR CNRS/Université Bretagne Sud, Vannes, France
flavio.oquendo@irisa.fr

[3] State University of Rio Grande do Norte, Natal, Brazil

Abstract. Software architecture description languages (ADLs) currently adopted by industry for software-intensive systems are largely semi-formal and essentially based on SysML and specialized profiles. Despite these ADLs allow describing both structure and behavior of the architecture, there is no guarantee regarding the satisfaction of correctness properties. Due to their nature, semi-formal ADLs do not support automated verification of the specified properties, in particular those related to safety and liveness of the specified behavior. This paper proposes a novel approach for empowering SysML-based ADLs with formal verification support founded on model checking. It presents (i) how the semantics of SysADL, a SysML-based ADL, can be formalized in terms of the CSP process calculus, (ii) how correctness properties can be formally specified in CSP, and (iii) how the FDR4 refinement checker allows verifying correctness properties through model checking. The automated model transformation from SysADL architecture descriptions to CSP composite processes has been implemented as a plug-in to the Eclipse-based SysADL Studio tool. This paper also describes an application of SysADL empowered with CSP to validate its usefulness in practice.

Keywords: Software architecture description · Formal verification · Correctness properties · CSP · SysML

1 Introduction

Software architecture descriptions play an essential role in the communication among stakeholders, e.g., architects, developers, etc. The precise communication of this artifact is quite important since a badly specified architectural model

This research was partially funded by INES 2.0, FACEPE grant APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0.

© Springer Nature Switzerland AG 2020
A. Jansen et al. (Eds.): ECSA 2020, LNCS 12292, pp. 101–117, 2020.
https://doi.org/10.1007/978-3-030-58923-3_7

causes design and implementation flaws in a software system and can create misunderstandings [8]. Architecture description languages (ADLs) have been used as means of expressing software architectures and producing models that can be used at design time and/or runtime [3].

One of the major challenges in the design of software-intensive systems consists in verifying the correctness of their software architectures, i.e., if the envisioned architecture is able to fully meet the established requirements. Acknowledged as an important activity in software industry [10,14], the architectural analysis aims to verify system properties using architectural models at design time to detect incorrectness, inconsistencies, and other undesirable issues as soon as possible in the software development process. Due to the critical nature of many complex software systems, rigorous architectural models (such as formal architecture descriptions) are quite desirable as means of better supporting automated architectural analysis. The main advantage of adopting a formal approach is precisely determining if a software system satisfies properties of interest and constraints related to requirements and check the accuracy and correctness of architectural designs. The literature indeed reports studies that combine formal verification and software architecture descriptions as means of ensuring safety, correctness, and consistency in software systems [1,18].

Despite describing structure and behavior of a software architecture is possible, there is no guarantee on its correctness properties. Some ways of validating if a software architecture was correctly designed with respect to its functionalities are generating source code in a given target programming language or producing executable models able to be simulated. Nonetheless, simulating the architecture neither constitutes a proof of satisfaction of safety and liveness properties nor a guarantee that the execution respects the specified architecture behavior. Another important concern is that semi-formal languages such as SysML have well-defined syntax, but they lack complete formal semantics. This hampers the automated verification of the specified properties, in particular those related to safety and liveness of the architecture behavior.

This paper presents an approach for empowering SysML-based architecture descriptions with formal verification to support the model checking of correctness properties. Such an approach relies on the Communicating Sequential Process (CSP) [15], a process calculus applied in both academia and industry to formally specify and verify the behavior of concurrent processes/systems and how they interact with each other. More specifically, this paper proposes a CSP-based semantics for SysADL [13], a SysML-based ADL that combines typical constructs of ADLs with the use of the popular diagrammatic notation based on the SysML Standard for modeling software-intensive systems. SysADL is aligned with the ISO/IEC/IEEE 42010 International Standard [7] for architectural descriptions by providing multiple viewpoints and views in terms of requirements, structure, behavior, and execution of software architectures.

The automated transformation model from SysADL architecture descriptions to CSP composite processes has been implemented and integrated into SysADL Studio [9], a free, open-source support tool for SysADL. The formal verifica-

tion itself is supported by FDR4 [5], a widely used refinement checker for CSP that allows verifying if the architectural model is free from deadlocks, livelocks, and miracles (i.e., specifications for which it is impossible to provide a valid implementation), as well as if the executable properties respect the behavioral specification. The application of SysADL formalized with CSP is herein illustrated with a room temperature control system (RTC).

The remainder of this paper is structured as follows. Section 2 briefly presents SysADL and CSP. Section 3 presents the CSP-based semantics for SysADL. Section 4 details the formal verification of properties regarding SysADL architecture descriptions with the FDR4 refinement checker. Section 5 presents the SysADL Studio extension to support the SysADL–CSP transformation. Section 6 discusses related work. Section 7 contains some concluding remarks.

2 Background

2.1 SysADL

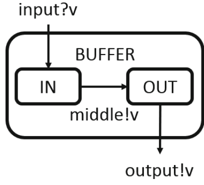
SysADL defines three software architecture viewpoints for a system, namely (i) structural, (ii) behavioral, and (iii) *executable*. The structural viewpoint defines the architectural elements composing the structure of a system (*components*, *ports*, *connectors*) and relationships among them. Communication among components takes place through connectors that bind input and output ports. SysADL requires declaring all elements before creating their instances. The elements are declared by using *Block Definition Diagrams* (BDDs) whereas the *Internal Block Diagram* (IBD) is used to specify how instances of components and connectors form the configuration of architectures.

The behavioral viewpoint details the behavior of (i) components and connectors through *activities*, *actions*, *constraints* and (ii) ports through *protocols*. Activity instances are described in the *Activity Diagram* by instantiating actions and flows. Activities or actions may have validation constraints specified through expressions in the OMG Action Language for Foundational UML (ALF). Constraints can be also expressed using the *Parametric Diagram*.

The executable viewpoint represents the concretization of both structural and behavioral viewpoints by simulating the architecture behavior at runtime. The main purpose of the simulation is validating the behavior logic regarding the satisfaction of requirements and analysis of architecture functionalities. In the executable viewpoint, it is possible to specify details of each action by using ALF statements as well as define and instantiate elements. The executable instances should be interpreted by an ALF engine to execute the architecture.

2.2 CSP

CSP is a process algebra that can be used to describe systems composed of independent, self-contained processes with interfaces to interact with the environment [15]. Most CSP tools such as FDR4 [5] and ProBE [4] accept a machine-processable CSP called CSP_M. For the sake of presentation, this paper uses the CSP notation in theoretical definitions and CSP_M in FDR4 verification assertions.



$N = 4$
datatype $ID = a \mid b$
channel $input, middle, output : ID$

$IN = input?v \rightarrow middle!v \rightarrow IN$
$OUT(s) = (\#s > 0)\ \&\ output!head(s) \rightarrow OUT(tail(s))$
$\qquad\qquad \Box\ (\#s < N)\ \&\ middle?v \rightarrow OUT(s \frown \langle v \rangle)$
$BUFFER = (IN \mathbin{[\![} \{\!|middle|\!\} \mathbin{]\!]} OUT(\langle\rangle)) \setminus \{\!|middle|\!\}$

Fig. 1. CSP specification of a bounded buffer.

The two basic CSP processes are *Stop* and *Skip*: the former does nothing, i.e., deadlocks, and the latter does nothing but terminates. Prefixing $a \rightarrow P$ is initially able to perform only the event a, afterwards it behaves as process P. Prefixing may have input or output values. Process $c?x \rightarrow P(x)$ assigns the received value c to the implicitly declared variable x and behaves as process P with x in scope. Process $c!e \rightarrow P(x)$ outputs the value c of the expression e and behaves as process P. A Boolean guard may be associated with a process: $g\ \&\ P$ behaves as process P if the predicate g is true, otherwise it deadlocks. The operator $P_1 \fatsemi P_2$ combines processes P_1 and P_2 in sequence. The external choice $P_1 \Box P_2$ initially offers events of both processes P_1 and P_2. The environment has no control over the internal choice $P_1 \sqcap P_2$, which is internally resolved. The sharing parallel composition $P_1 \mathbin{[\![}\ cs\ \mathbin{]\!]} P_2$ synchronizes processes P_1 and P_2 on events in the synchronization set cs, so that events that are not listed occur independently. Processes composed in interleaving $P_1 \mathbin{|||} P_2$ run independently. The event hiding operator $P \setminus cs$ encapsulates events in cs.

Figure 1 illustrates the CSP specification of a bounded buffer. There are two declarations: N is a constant with value 4 and ID is a datatype whose values are a and b. Process IN is the buffer component that receives a value through channel *input* and sends it to process OUT via channel *middle*. As process OUT can store N elements, it may receive new values via channel *middle* if the size of its sequence has not reached its capacity ($\#s < N$). The received value is stored at the tail of its sequence ($s \frown \langle v \rangle$). Process OUT may also provide an *output*, but only if its sequence is not empty ($\#s > 0$). In this case, it writes the head of the sequence ($head(s)$) and keeps only its tail ($tail(s)$). $BUFFER$ is the parallel composition of process instances of IN and OUT starting with the empty sequence. Both processes IN and OUT synchronize on channel *middle*, which is hidden from the environment.

3 A CSP-Based Formal Semantics for SysADL

The translation from a SysADL architectural description to a CSP-based formal semantics allows verifying properties such as deadlock-freedom, livelock-freedom, and consistency among the structural, behavioral, and execution viewpoints of the model (see Sect. 4). The translation of types, those viewpoints, and the

overall model are herein presented by using a room temperature control (RTC) system as running example[1]. The RTC system uses two temperature sensors to capture the current temperature. A user can set the desired temperature. A central controller receives the values from temperature sensors, compares them with the desired temperature, and turns the cooler/heater on/off. The system has a motion sensor to detect if there is someone in the room. In case of presence, the system operates to provide the desired temperature, otherwise it operates to keep the temperature at 22 °C.

Fig. 2. Enumeration and composite datatypes in SysADL (left) and their translation to CSP (right).

Types. SysADL types are used in different viewpoints of the architecture description. These types can be basic types (*Integer*, *Boolean*, *String*, and *Real*), enumerated types, and composite types resulted from the composition of other types. *Integer* and *Boolean* are respectively mapped to *Int* and *Bool*. The *String* type is mapped to *String*, a set containing finite sequences of CSP characters (*Char*) with a maximum length. *Real* is translated to a pair of *Int* values. Enumerated types are mapped to CSP datatypes, which allow defining new types along with an enumeration of its values. Composite types are mapped to sets of tuples whose values come from the basic types.

Figure 2 shows the SysADL *Command* type and its mapping to a CSP datatype. A variable of the *Command* enumerated type may assume only values *On* or *Off*. The *Commands* type is a composition of two values of the *Command* type (*heater* and *cooler*). The *Commands* type is translated to a set definition that declares a set of pairs of values of the *Command* type.

3.1 Structural Viewpoint

Ports. SysADL ports are interaction points between a component and other architectural elements. They represent how data flow from a component (*out* ports) to another component (*in* ports). Composite ports are composed of other ports. Figure 3 presents an example of a port definition in SysADL. *CTemperatureITP* is an input port through which data of the *CelsiusTemperature* type flow.

Previously defined ports can be instantiated in component definitions, which can themselves be instantiated in configurations. In the CSP semantics for

[1] CSP files and the extended SysADL Studio are available at http://bit.ly/2PAqYiD.

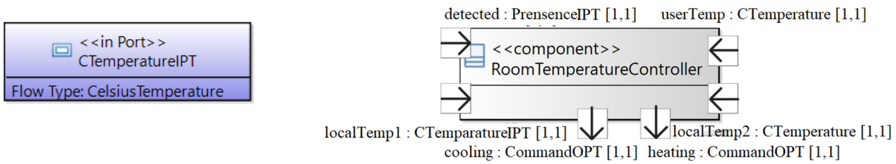


Fig. 3. Example of port (left) and composite component (right) in SysADL.

SysADL, a channel is declared for each port instantiated in component definitions or component instantiations. A simple port leads to the declaration of a CSP channel and a composite port leads to the declaration of one CSP channel for each port composing it. The names of the CSP channels related to ports attached to component definitions contain the name of the port and the name of its definition whereas the name of the channels related to component instantiations also includes the name of the instance. In Fig. 3, the instantiation *localTemp*1 of the port *CTemperatureITP* in the composite component *RoomTemperatureControllerCP* and the instantiation *rtc* of this component respectively yield the following CSP declarations:

channel $localTemp1_CTemperatureIPT$: $CelsiusTemperature$
channel $rtc_localTemp1_CTemperatureIPT$: $CelsiusTemperature$

Components. SysADL components can be defined as either (i) boundary components, i.e., they interface and exchange data with the physical environment or (ii) non-boundary components. These components repeatedly receive all input data through input ports, process them, and provide all their outputs in output ports. In Fig. 3, *RoomTemperatureControllerCP* is defined as a non-boundary component that receives four inputs and provides two outputs.

Figure 4 presents the architectural configuration of the RTC system in SysADL. This architecture is composed of seven component instances. $s1$, $s2$ and $s3$ are sensors that collect data about temperature and presence of people at the monitored environment. These data are processed by component *rtc*, which plays the role of the room temperature controller. Actuators $a1$ and $a2$ control cooling and heating according to decisions taken by *rtc*. *ui* is a user-interface component. In the translation of the definition and instantiation of these components, simple components yield processes whose behavior is the process that translates its activity, whilst composite components yield processes whose behavior is the process that translates its configuration. In Fig. 5, the simple component *PresenceCheckerCP* is translated to *PresenceCheckerCP* = *CheckPresenceToSetTemperatureAC* and the composite component *RoomTemperatureControllerCP* is translated to *RoomTemperatureControllerCP* = *RoomTemperatureControllerCP_Config*.

The semantics of boundary components such as *TemperatureSensorCP* (see Fig. 5) considers their non-deterministic behavior. For this reason, their semantics differs from that given for non-boundary components: the resulting process

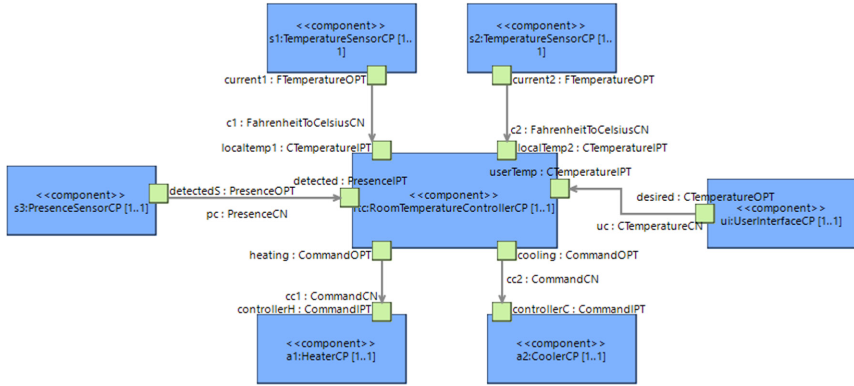


Fig. 4. Configuration of the RTC system in SysADL.

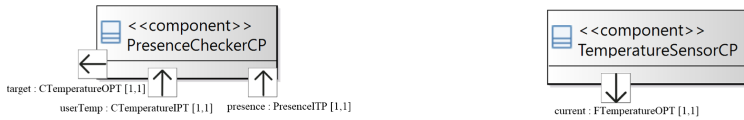


Fig. 5. Examples of SysADL boundary components.

randomly chooses a value to communicate in their output ports. As an example, the semantics of *TemperatureSensorCP* is a process that non-deterministically chooses a value from the type of the *FahrenheitTemperature* port.

Connectors. SysADL connectors bind ports of the connected components for exchanging data, possibly with some processing during transmission. In the CSP semantics for SysADL, connectors that do not process data are represented as CSP processes. Once connected to an output port and an input port, these processes repeatedly receive values from the output port and write them to the input ports. The behavior of a one-place buffer allows assynchronous communications among components, exactly corresponding to communications among components in SysADL. For example, the connector *CommandCN* (see Fig. 4) is translated to the following CSP process:

$$\begin{aligned} CommandCN = \ & commandOut_CommandOPT?out \rightarrow \\ & commandIn_CommandIPT!out \rightarrow CommandCN \end{aligned}$$

Connectors that process data have their behavior defined by activities and their translation follows the same approach. For instance, *FahrenheitToCelsiusCN* is a connector that receives a Fahrenheit temperature and outputs the corresponding Celsius temperature. This conversion is defined by process *FahrenheitToCelsiusAC* as the behavior of the connector (see Fig. 6).

Configuration. In the SysADL structural viewpoint, the configuration defines how component instances are connected by connector instances. The behavior of a configuration *CFD* is the parallel composition of all its compo-

Fig. 6. Definition of a connector in SysADL (left) and its translation to CSP (right).

nents ($Components_C FD$) and connectors ($Connectors_C FD$) synchronizing on the channels that correspond to the ports ($Sync_C FD$). Internal ports of the configuration ($Internal_C FD$) are hidden. For example, the configuration of the RTC system presented in Fig. 4 is translated to the following definitions:

$$
\begin{aligned}
&RTCSystemCFD_config = \\
&\quad \left(\begin{array}{l} Components_RTCSystemCFD \\ [\![Sync_RTCSystemCFD]\!] \\ Connectors_RTCSystemCFD \end{array} \right) \setminus Internal_RTCSystemCFD \\
&Sync_RTCSystemCFD = \\
&\quad \{\!| current_FTemperatureOPT, current_FTemperatureOPT, \\
&\quad\; detected_PresenceOPT, desired_CTemperatureOPT, \\
&\quad\; controllerC_CommandIPT, controllerH_CommandIPT |\!\} \\
&Internal_RTCSystemCFD = \\
&\quad \{\!| detectedRTC_PresenceIPT, heatingRTC_CommmandOPT, \\
&\quad\; coolingRTC_CommandOPT, userTempRTC_CTemperatureIPT, \\
&\quad\; localTemP1_CTemperatureIPT, localTemP2_CTemperatureIPT |\!\}
\end{aligned}
$$

The processes corresponding to components and connectors of the configuration are defined as the interleaving of all components and connectors instances. The instantiation is achieved by using CSP renaming: every channel is renamed to a channel prefixed with the instance name and using the port instantiation name, rather than the port name. The resulting CSP specification[2] would be:

$$
\begin{aligned}
&Components_RTCSystemCFD = \\
&\quad ||| \; TemperatureSensorCP \\
&\qquad [current_FTemperatureOPT \leftarrow s1_current1_FTemperatureOPT] \\
&\quad ||| \; TemperatureSensorCP \\
&\qquad [current_FTemperatureOPT \leftarrow s2_current2_FTemperatureOPT] \\
&\quad ||| \; PresenceSensorCP[\ldots] \; ||| \; UserInterfaceCP[\ldots] \; ||| \; CoolerCP[\ldots] \\
&\quad ||| \; HeaterCP[\ldots] \\
&\quad ||| \; RoomTemperatureControllerCP[\ldots]
\end{aligned}
$$

$$
\begin{aligned}
&Connectors_RTCSystemCFD = \\
&\quad FahrenheitToCelsiusCN \\
&\qquad \left[\begin{array}{l} Ct_CTemperatureIPT \leftarrow rtc_localtemp1_CTemperatureIPT, \\ Ft_FTemperatureOPT \leftarrow s1_current1_FTemperatureOP \end{array} \right] \\
&\quad ||| \; FahrenheitToCelsiusCN[\ldots] \; ||| \; DetectedCN[\ldots] \\
&\quad ||| \; CTemperatureCN[\ldots] \\
&\quad ||| \; ControlCommandCN[\ldots] \; ||| \; ControlCommandCN[\ldots]
\end{aligned}
$$

[2] For the sake of conciseness, parts of the specification are omitted. The complete version can be found at http://bit.ly/2PAqYiD.

The proposed translation approach is indeed compositional. Therefore, the translation of simple and composite components follow the same rules. For instance, the configuration of the composite component *RoomTemperatureControllerCP* presented in Fig. 7 is translated like *RTCSystemCFD_config*, i.e., the parallel composition of its component and connector instances synchronizing on the channels that correspond to the ports with its internal events hidden from the environment. However, there is a minor increment in the definition of the process that represents connectors, *Connectors_RoomTemperatureControllerCP*: it also interleaves a process that translates the delegations, which are special connectors between proxy ports and ports in components as presented in Fig. 7:

$$\begin{aligned}&Connectors_RoomTemperatureControllerCP =\\&\quad CTemperatureCN[\ldots] \mid\mid\mid CTemperatureCN[\ldots] \mid\mid\mid Delegation_rtc\\&Delegation_rtc = detectedRTC_to_detected \mid\mid\mid userTempRTC_to_userTemp\\&\qquad \mid\mid\mid localtemp1_to_s1 \mid\mid\mid localTemp2_to_s2\\&\qquad \mid\mid\mid heating_to_heatingRTC \mid\mid\mid cooling_to_coolingRTC\end{aligned}$$

For illustration purposes, the translation of delegation *detectedRTC_to_detected* is presented in the following. As for connectors, the behavior of a one-place buffer allows assynchronous communications among components, exactly corresponding to the behavior of SysADL delegations. The other translations follow the same approach.

$$\begin{aligned}&detectedRTC_to_detected =\\&\quad rtc_detected_PresenceIPT?PresenceIPT \rightarrow\\&\quad pc_detected_PresenceIPT!PresenceIPT \rightarrow detectedRTC_to_detected\end{aligned}$$

3.2 Behavioral Viewpoint

In SysADL, the behavioral viewpoint defines the behavior of components, connectors, and ports of the model. The behavior is described in terms of activities, actions, and constraints.

Constraints. Constraints are described as predicates that can be used to restrict the set of values of an activity. Once defined, constraints can be used in actions. As an example, Fig. 8 presents the constraint *FahrenheitToCelsiusEQ*, which verifies if the values given as arguments correctly correspond to the same temperatures in both Celsius and Fahrenheit units, and its translation to the CSP function $FahrenheitToCelsiusEQ(f, c)$.

Actions. SysADL actions process arguments given as inputs and provide an output that must respect its constraints. As many possible outputs may exist for the same input, the translation considers a non-deterministic choice of such possible output values. For example, the action *FahrenheitToCelsiusAN* returns the temperature value in the Celsius unit that corresponds to the temperature given in the Fahrenheit unit. In the translation, this corresponds to a communication on the channel that represents the output port. The translation of *FahrenheitToCelsiusAN* called by the connector named $s1$ is presented in Fig. 8.

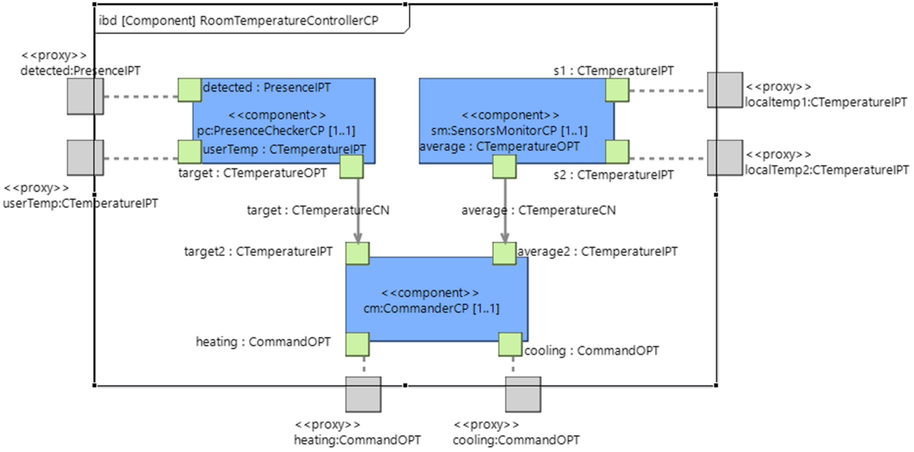
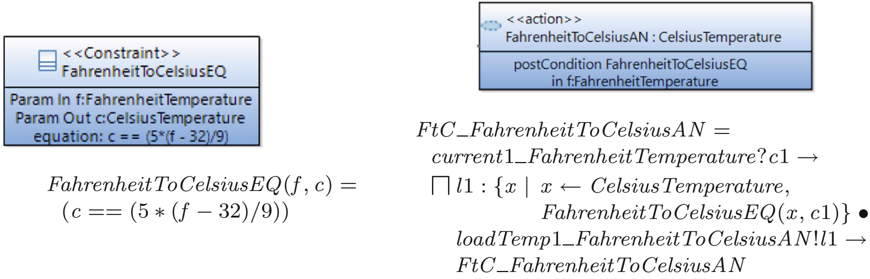


Fig. 7. Configuration of a composite component in SysADL.



$$FahrenheitToCelsiusEQ(f, c) = (c == (5 * (f - 32)/9))$$

$$FtC_FahrenheitToCelsiusAN = current1_FahrenheitTemperature?c1 \rightarrow \sqcap l1 : \{x \mid x \leftarrow CelsiusTemperature, FahrenheitToCelsiusEQ(x, c1)\} \bullet loadTemp1_FahrenheitToCelsiusAN!l1 \rightarrow FtC_FahrenheitToCelsiusAN$$

Fig. 8. Examples of constraint (left) and action (right) in SysADL.

Activities. SysADL activities are composed of one or more actions, which may communicate values between them. Figure 9 shows the *DecideCommandAC* activity as the composition of three actions that communicate values among them: actions *CommandHeaterAN* and *CommandCoolerAN* receive the output of the *CompareTemperatureAN* action.

The result of the translation of both constraints and actions is used in the translation of activities. Similarly to the translation of actions, the translation of activities also takes the name of the allocated component or connector. The activity is translated to a parallel composition of processes considering the activity entry and exit points (pins), i.e., the allocation of the activity on the associated component. For example, the activity *DecideCommandAC* is composed of actions *CompareTemperatureAN*, *CommandHeaterAN*, and *CommandCoolerAN*. The translation of this activity is:

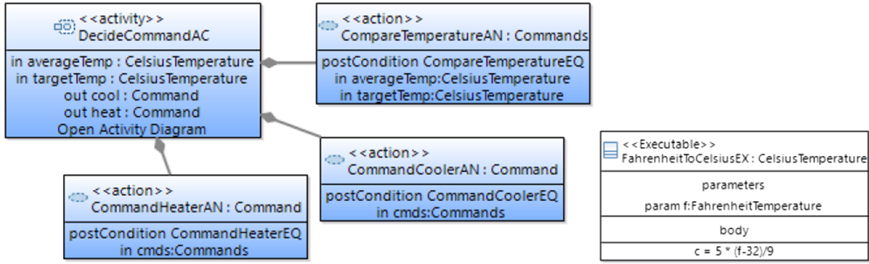


Fig. 9. Examples of activity (left) and executable elements (right) in SysADL.

$$DecideCommandAC = \begin{pmatrix} Pins_DecideCommandAC \\ [\![Sync_DecideCommandAC]\!] \\ Actions_DecideCommandAC \end{pmatrix} \\ \setminus Internal_DecideCommandAC$$

Process $Pins_DecideCommandAC$ is the parallel composition of the processes that represent all pins in the activity. The pin $average2_Decide$ $CommandAC$ receives the value from the component port $average2$ and sends it to the $DecideCommandAC$ activity pin.

$$\begin{array}{l} Pins_DecideCommandAC = \\ \quad [\![Sync_DecideCommandAC]\!]\ i : \{1\,..\,4\} \bullet \\ \quad\quad Pins_DecideCommandAC_Func(i) \\ Pins_DecideCommandAC_Func(1) = average2_DecideCommandAC \\ \ldots \\ Pins_DecideCommandAC_Func(4) = cooling_DecideCommandAC \\ average2_DecideCommandAC = average2_CTemperatureIPT?average2 \rightarrow \\ \quad\quad average2_CelsiusTemperature!average2 \rightarrow \\ \quad\quad average2_DecideCommandAC \\ \ldots \end{array}$$

Similarly, process $Actions_DecideCommandAC$ is the parallel composition of the processes that represent all actions in the activity:

$$\begin{array}{l} Actions_DecideCommandAC = \\ \quad [\![Sync_DecideCommandAC]\!]\ i : \{1\,..\,3\} \bullet \\ \quad\quad Actions_DecideCommandAC_Func(i) \\ Actions_DecideCommandAC_Func(1) = ct_CompareTemperatureAN \\ \ldots \end{array}$$

3.3 Execution Viewpoint

The execution viewpoint must satisfy the conditions defined in the actions and related constraints within the behavioral viewpoint. The translation of executable elements translates their bodies to CSP functions. For example, the

FahrenheitToCelsiuEX executable element (see Fig. 9) receives a temperature value in the Fahrenheit unit and returns another one in the Celsius unit. The translation of the executable element *FahrenheitToCelsiuEX* is a CSP function parameterized on the temperature value in the Fahrenheit unit according to the equation $FahrenheitToCelsiuEX(f) = (5 * (f - 32)/9)$.

4 Formal Verification of SysADL Models

The translation of the SysADL models to CSP fosters their formal verification. This work uses FDR4 [5], a refinement model checker for CSP to automatically verify if the model satisfies (i) deadlock-freedom, (ii) livelock-freedom, (iii) absence of miracles, and (iv) the compliance of the execution model with the behavioral model. The translation of the model and the verification of these properties are fully automatic. The compliance with functional requirements can also be automatically verified. Nevertheless, the specification of the requirement currently needs to be expressed in CSP. The implementation of a user-friendly functional requirement description UI using SysADL diagrams is underway.

Deadlock-freedom and livelock-freedom are classical concurrency properties. A deadlock happens when a group of processes are permanently held on a situation in which each process waits for resources held by another process in the group. This makes the process to not progress. A livelock also has the same consequence, but for a different reason. In a livelock, processes are indefinitely progressing with internal events that cannot be seen by the external environment. This absence of external event leads the system to present no progress. It is possible to easily check the resulting CSP processes against these two properties by using FDR4 standard assertions for deadlock- and livelock-freedom. For example, the following assertions can be used in FDR4 to check if the running example (modeled as process *RTCSystemCFD*) is free of deadlock and livelock:

```
assert RTCSystemCFD:[deadlock free]
assert RTCSystemCFD:[divergence free]
```

Another property to be verified is that the behavioral model is not a miracle, i.e., the model has no possible executable model. Considering a SysADL constraint C defined in terms of inputs $i_1 \dots i_n$ and outputs $o_1 \dots o_m$, for every possible combination of input values that satisfy the pre-condition constraint *pre*, there must exist output values satisfying the post-condition constraint *post*. Formally, it is defined a CSP process that diverges if, and only if, the constraint is a miracle. For this verification, an auxiliary process $IS_TRUE(c)$ is defined as successfully terminating only if predicate c is true, otherwise it diverges.

The process created for each constraint receives all input values and checks the pre-condition by using a guarded process $pre \ \& \ IS_TRUE(\dots)$. If *pre* is false, then the process deadlocks avoiding a divergence, otherwise it checks if a set defined using the CSP set comprehension notation is not empty. This set contains all tuples $(o_1, \dots, o_m)$ with values $o_1, \dots, o_m$ respectively are of type $T_1, \dots, T_n$ and satisfy the post-condition $post(i_1, \dots, i_n, o_1, \dots, o_m)$. Informally,

this set is not empty if, and only if, the constraint is not a miracle. Therefore, it is possible to find output values satisfying the constraints when the input values satisfy the pre-condition.

```
C_check = C_i1?i1 -> ... C_in?in ->
          pre(i1,...,in) &
             let S = {(o1,...,om) | o1 <- T1, ..., om <- Tm,
                                    post(i1,...,in,o1,...,om)}
             within IS_TRUE(not(S == {}))
assert C_check:[livelock free]
```

In the running example, the constraint *FahrenheitToCelsiusEQ* is verified against miracles with the following assertion:

```
FahrenheitToCelsiusEQ_check =
  FahrenheitToCelsiusEQ_f?f ->
  true & let S = {c | c <- CelsiusTemperature, FahrenheitToCelsiusEQ(f,c)}
         within IS_TRUE(not(S == {}))
assert FahrenheitToCelsiusEQ_check:[livelock free]
```

The last verification is that the execution model is a refinement of the behavioral model. The only difference between these models regards the specification of actions, which are replaced by their executions. The former is composed of possibly constrained actions whereas the latter provides procedures that implement the behavior specified in the actions. This implementation must respect the constraints described in the activity. Theorem 1 states that an indexed internal choice over a set S is a failures-divergences refinement of an indexed internal choice over a set T if, and only if, S is a subset of T.

Theorem 1. $\sqcap x : T \bullet P(x) \sqsubseteq_{FD} \sqcap x : S \bullet P(x) \Leftrightarrow S \subseteq T$

The verification if the execution is a refinement of the behavior is done by simply checking subset containment: the set of pairs satisfying the executable *FahrenheitToCelsiusEX* must be a subset of the set of pairs satisfying *FahrenheitToCelsiusEQ*. Current work includes the integration with the CVC4 SAT solver [2] to optimize this verification.

$$
\begin{aligned}
\mathit{FahrenheitToCelsiusEQ_s} = \{(f, c) \mid\ & f \leftarrow \mathit{FahrenheitTemperature}, \\
& c \leftarrow \mathit{CelsiusTemperature}, \\
& \mathit{FahrenheitToCelsiusEQ}(f, c)\} \\
\mathit{FahrenheitToCelsiusEX_s} = \{(f, c) \mid\ & f \leftarrow \mathit{FahrenheitTemperature}, \\
& c \leftarrow \mathit{CelsiusTemperature}, \\
& c == \mathit{FahrenheitToCelsiusEX}(f)\}
\end{aligned}
$$

```
assert IS_TRUE(subset(FahrenheitToCelsiusEX_s,
                      FahrenheitToCelsiusEQ_s)):[divergence free]
```

5 Tool Support and Validation

The translation from SysADL architectural models to CSP processes and the verification of the resulting processes has been implemented as a plug-in to the Eclipse-based SysADL Studio tool [9]. The main thrust behind implementing the plug-in is making both formalization and verification as much transparent as possible to end-users. The translation to CSP and further verification of SysADL models require a single action: the user selects the SysADL model and then the verification operation. This action opens a window at which the user can select the configuration of the SysADL model to be verified.

The tool translates[3] the SysADL model to CSP by using the rules presented in Sect. 3, interacts with FDR4, analyzes the verification results from this interaction, and presents them in a user-friendly way. For each verified property, the tool shows whether it has been satisfied or not. When the property has not been satisfied, a trace that exemplifies the violation of the property is textually displayed. Current work also includes visually displaying the indication of the problem source at the SysADL diagram itself.

The validation of the correctness and effectiveness of the proposed approach and tool support consisted in using the plug-in to verify the aforementioned properties in existing SysADL models publicly available at the literature[4]. These properties were also verified in variations of such models as means of intentionally inserting errors and confirming that the proposed approach indeed identify them. As an example, a miraculous specification was identified in the model presented in Sect. 3. The original authors used the same range of natural numbers for temperature values in Celsius and Fahrenheit units, thus making it impossible to find valid values in the Celsius unit to every valid value in the Fahrenheit unit while respecting the equation $celsius = (5 * (fahrenheit - 32)/9)$.

The errors intentionally inserted into the original models were also successfully identified. For instance, the implementation of *FahrenheitToCelsiusEX* in the execution model was changed to $celsius = 5 * (fahrenheit + 32)/9$ and the plug-in identified that the execution model has not respected the specification of the behavioral model.

The implemented plug-in was also able to successfully verify the compliance of SysADL architectural models with functional requirements. A first requirement was that the cooler and the heater cannot be turned on at the same time (safety property). Another requirement was that if no presence is detected in the room, then its temperature is always adjusted to a predefined temperature (liveness property). These requirements are currently expressed as CSP processes. Future work will address the description of such requirements by using SysADL diagrams.

Ongoing work also includes computational experiments to demonstrate the scalability of the proposed approach. Preliminary results obtained with the run-

[3] The translation is implemented in Acceleo (http://www.eclipse.org/acceleo/).

[4] Available at http://sysadl.org.

ning example[5] showed a overall time of 774 ms when performing the verification on a computer with an Intel® Core™ i5 processor, 8 GM of RAM, and Microsoft® Windows 10 as operating system. These results demonstrated a linear increase of the verification time with the number of instances.

6 Related Work

The literature reports approaches with formal verification of software architectures based on model checking [1,18]. Some of them have formalized their architectural descriptions as one of the primary means of ensuring reliability, security, correctness, and consistency of their projects. However, as far as it is known, none of them targets improving a SysML-based ADL with formal verification founded on model checking and with tool support. This is specially interesting for industry, which largely adopts SysML-based modeling languages [10].

Mouraditis et al. [12] defined a set of structural, behavioral, and security primitives and conceptualized it with the Z specification language to capture a core architectural model to build secure architectures. The approach herein proposed does not rely on a restricted set of architectures, but rather on any software architecture modeled using SysADL, which is a general-purpose ADL. The Mokni et al.'s work [11] considered software architecture changes to be verified and validated as means of ensuring a valid, reliable evolution process. The authors proposed a set of rules defined as a B formal model of the Dedal ADL along with consistency properties, which were checked and validated by using the ProB animator and model checker. The approach proposed here also ensures consistency among different elements of a SysADL model, but it focuses on the consistency among different viewpoints and takes advantage of the use of a process algebra (CSP) rather than a model-based formalism (B or Z) to guarantee concurrency aspects of the model, such as the safe interaction among components (deadlock- and livelock-freedom). The Taoufik et al.'s work [17] proposed to open UML 2.0 on the Wright ADL to verify the behavioral consistency of architectures. The compatibility with the Wr2Fdr tool [16] motivated the use of Wright/CSP since the tool generates eleven standard properties related software architecture consistency. Moreover, the Wright/CSP target configuration can be automatically translated to an FDR specification acceptable by the FDR2 model-checker. Besides providing SysADL with the same verification possibilities, the proposed approach allows verifying concurrency properties and functional requirements. Furthermore, SysADL Studio was integrated with the translator to CSP and its communication with FDR4 in a transparent way to users.

7 Conclusion

This paper presented a CSP-based approach to support the automated formal verification of properties specified in SysADL, a semi-formal SysML-based ADL.

[5] A short demo is available at https://youtu.be/vlchTK3fk2Y.

The solution relies on empowering SysADL with model checking by combining the CSP process algebra and the FDR4 model checker, besides providing a semantics for SysADL diagrams. With the proposed approach, it was possible to verify properties related to deadlocks, livelocks, miracles, and consistency among the different viewpoints of the specified configuration-based behavior. The concretization of the approach in the Eclipse-based SysADL Studio tool allowed validating it in several scenarios, including the example presented throughout this paper. The same approach can be applied to other SysML-based ADLs to formally verify architectural properties. Future work includes providing a π-calculus based semantics for SysADL and the formalization of both translations and a cross-verification of the semantics by using the strategy presented in the Unifying Theories of Programming (UTP) [6].

References

1. Araujo, C., Cavalcante, E., Batista, T., Oliveira, M., Oquendo, F.: A research landscape on formal verification of software architecture description. IEEE Access **7**, 171752–171764 (2019)
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Clements, P., et al.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Reading (2011)
4. Formal Systems (Europe) Ltd.: Process Behaviour Explorer - ProBE User Manual. FSEL, United Kingdom (2003)
5. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3: a parallel refinement checker for CSP. Int. J. Softw. Tools Technol. Transfer. **18**, 149–167 (2016)
6. Hayes, I.J., Meinicke, L.A.: Developing an algebra for rely/guarantee concurrency: design decisions and challenges. In: Ribeiro, P., Sampaio, A. (eds.) UTP 2019. LNCS, vol. 11885, pp. 176–197. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31038-7_9
7. ISO/IEC/IEEE 42010: Systems and Software Engineering - Architecture Description. ISO, Switzerland (2011)
8. Lago, P., Malavolta, I., Muccini, H., Pelliccione, P., Tang, A.: The role ahead for architectural languages. IEEE Softw. **32**(1), 98–105 (2015)
9. Leite, J., Batista, T., Oquendo, F., Silva, E., Santos, L., Cortez, V.: Designing and executing software architectures models using SysADL Studio. In: Proceedings of the 2018 IEEE International Conference on Software Architecture Companion, USA, pp. 81–84. IEEE (2018)
10. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Software Eng. **39**(6), 869–891 (2013)
11. Mokni, A., Huchard, M., Urtado, C., Vauttier, S., Zhang, H.Y.: Formal rules for reliable component-based architecture evolution. In: Lanese, I., Madelaine, E. (eds.) FACS 2014. LNCS, vol. 8997, pp. 127–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15317-9_8

12. Mouratidis, H., Kolp, M., Faulkner, S., Giorgini, P.: A secure architectural description language for agent systems. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 578–585. ACM, New York (2005)
13. Oquendo, F., Leite, J., Batista, T.: Software Architecture in Action: Designing and Executing Architectural Models with SysADL Grounded on the OMG SysML Standard. Springer, Switzerland (2016). https://doi.org/10.1007/978-3-319-44339-310.1007/978-3-319-44339-3
14. Ozkaya, M.: Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling? Inf. Softw. Technol. **95**, 15–33 (2018)
15. Roscoe, A.W.: Understanding Concurrent Systems. Springer, London (2010). https://doi.org/10.1007/978-1-84882-258-0
16. Rouis, T.S., et al.: Wr2Fdr tool maintenance for models checking. In: Fujita, H., Selamat, A., Omatu, S. (eds.) New Trends in Intelligent Software Methodologies, Tools and Techniques, Frontiers in Artificial Intelligence and Applications, vol. 297, pp. 425–440. IOS Press, Amsterdam (2017)
17. Taoufik, S.R., Tahar, B.M., Mourad, K.: Behavioral verification of UML2.0 software architecture. In: Proceedings of the 12th International Conference on Semantics, Knowledge and Grids, pp. 115–120 (2016)
18. Zhang, P., Muccini, H., Li, B.: A classification and comparison of model checking software architecture techniques. J. Syst. Softw. **83**(5), 723–744 (2010)