Compile-time dynamic and recursive data structures in Modelica

[Work in Progress]

Matthias Hellerer
German Aerospace Center (DLR)
Institute of System Dynamics and Control
Oberpfaffenhofen, Germany
Matthias.Hellerer@DLR.de

Fabian Buse
German Aerospace Center (DLR)
Institute of System Dynamics and Control
Oberpfaffenhofen, Germany
Fabian.Buse@DLR.de

ABSTRACT

The current Modelica Standard (v3.3) does not support dynamic or recursive data structures. For many applications this constitutes a serious restriction rendering certain implementations either impossible or requires elaborate and unelegant constructs. In this paper we will show that support for dynamic and recursive data structures can be implemented in the Modelica IDE Dymola using a variety of advanced constructs. This proves the principle viability of the then proposed inclusion of those data structures in the Modelica Standard.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; *Simulation languages*; • **Theory of computation →** *Data structures design and analysis*;

KEYWORDS

Modelica, Dymola, dynamic data structures, recursive data structures, language enhancement

ACM Reference Format:

Matthias Hellerer and Fabian Buse. 2017. Compile-time dynamic and recursive data structures in Modelica: [Work in Progress]. In *EOOLT'17: 8th International Workshop on Equation-Based Object-Oriented Languages and Tools, December 1, 2017, Wessling, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3158191.3158205

1 INTRODUCTION

Data structures in Modelica are generally static. They are pre-allocated during compilation and cannot be changed later, as this would necessitate a structural change of the model. Dynamic data structures are data elements which can change their structure. For the purpose of this paper we will introduce a differentiation between run-time dynamic data structures and compile-time dynamic data structures. In the context of Modelica and similar modeling languages, run-time dynamic data structures would be able to change their structure while the simulation is performed and therefore in reaction to simulation results. For example adding a new value to an array every time a certain event occurs[1]. This is traditionally what is referred to as dynamic data structures. Compile-time dynamic structures similarly may change their structure but only during the compilation process and are static afterwards, during run-time [3].

A Modelica tool forms a models differential-algebraic equation (DAE) system during compilation and it cannot be changed during run-time. Attempts to change this are still actively researched but not yet available [2, 5, 14, 15]. None the less, before the DAE system is formed, dynamic data structures can be used as will be shown here. A somewhat similar concept is already part of Modelica, evenso with a somewhat limited scope, in the form of *expendable connectors* (Often refered to as *busses*.). An *expendable connector* allows the user to add arbitrary values to a connection after the connector was defined. Here the values on the *bus* are also determined at compile-time [1].

Recursive data structures are data structures containing data of their own type[11]. A very common construct in computer science and pre-requisit for many applications. For example linked lists are comprised of a data elements and either the next element itself or a link (e.g. a pointer) to the next element in the list, which itself shows the same structure [9, 13]. Listing 1 demonstrates a linked list in Modelica syntax.

Listing 1: Simple linked list

```
1 record linkedListElement
2   Real data;
3   linkedListElement nextElement;
4 end linkedListElement;
```

This is repeated for all elements of the list until its end is indicated (e.g. by a `nullptr`) [10]. Such data structures are currently not supported by Modelica (as of v3.3) [1]. A Modelica record may not contain a reference to itself and has no concept of pointers. We will show here how compile-time dynamic arrays may be used similar to pointers, allowing us to implement recursive data structures.

This proves the principle viability of compile-time dynamic data structures in Modelica and show-cases their usefulness. We therefore propose the inclusion of those data structures in the Modelica Standard in the end. All libraries and applications demonstrated here are currently only tested with the Modelica IDE Dymola by Dassault Systèmes [4].

2 UNIQUE IDENTIFIER (UIDS) LIBRARY

This whole project started with a simple library. Often times `ExternalObjects` require an unique identifier (UID). In large projects and when using replaceable objects, tracking all instances of such an

[1] really adding a new value to the array, not changing a pre-existing array value

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EOOLT'17, December 1, 2017, Wessling, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-6373-0/17/12…$15.00
https://doi.org/10.1145/3158191.3158205

external object and assigning unique IDs can be tedious and error prone. We therefore developed a small library using an inner/outer construct which allowed us to automatically assign unique IDs. The library later grew to incorporate multiple groups (e.g. when using multiple C-libraries in one model) within which the IDs had to be unique and a counter for the total number of IDs within such a group was added.

Based on this library we started to experiment and soon discovered that UIDs could not only be used for external objects but also in Modelica code, yet only in very simple models. While C-code will almost always accept the UID at run-time, most interesting applications in Modelica require the UID value at compile-time. That is before the model is flattened and the DAE is formed. For example accessing a value in an array using UIDs works, but using this value then in an equation and incorporating it into the DAE system, requires the compiler to know which value is used. The problem is, that the UID value is also determined during the compilation. It is therefore not possible to use an UID in this manner. To circumvent this problem we found that is possible to write certain information to a file before the compilation process itself is started using `final parameter`, `algorithms` as well as the `__Dymola_preInstantiate` command[2]. The file may then be accessed to retrieve the IDs during compilation.

Listing 2: UID Library models

```
1  model UniqueID
2    parameter String group = "none";
3    final parameter Integer uid = getUID(group);
4    annotation (__Dymola_preInstantiate =
5      registerUID(group));
6  end UniqueID;
7
8  model GroupTotal
9    parameter String group = "none";
10   final parameter Integer total = getTotal(group);
11 end GroupTotal;
```

The UID Library consists of two main models: `UniqueID` and `GroupTotal` as outlined in Listing 2. `UniqueID`-objects are assigned a group and they provide an integer value `uid`, unique within this group. The `uid` values start at 0, due to their emergence from the *C*-interface and are in the range $[0..total[$. `GroupTotal`-objects similarly provide the total number of values assigned within a certain group.

3 COMPILE-TIME DYNAMIC ARRAYS

UIDs can not only be used with `ExternalObjects` but also in Modelica code where their compile-time availability allows them to be used similar to `constants`. Specifically they can be used to not only access values in an array but also to determine the size of the array. For an example see Listing 3.

Listing 3: Compile-time dynamic array in Modelica

```
1  model ArrayTest
2    UID.UniqueID id1;
3    UID.UniqueID id2;
4    UID.GroupTotal gtotal;
5    Real array[gtotal.total];
6    equation
7    array[id1.uid+1] = 1;
8    array[id2.uid+1] = 5;
9  end ArrayTest;
```

[2]This non-standard command currently limits our implementation to Dymola IDE

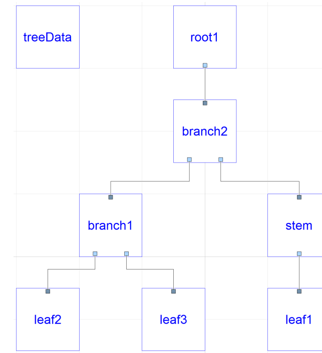

Figure 1: A simple data tree in Modelica

First two `UniqueID`-objects and one `GroupTotal`-object are created (by default they will be assigned the group name *"none"*). The total can then be used to create an array of fixed size `gtotal.total` (line 5). Furthermore the `UniqueIDs` can be used to identify objects in this array and use them in Modelica code (line 7).

In this example all objects are easily visible in one model and the user could edit it manually, but of course this could be a lot more complicated and spread over many objects in a real application. Thereby making the array size less obvious and requiring the user to keep track of used IDs. Furthermore this can easily get even more complicated with `replaceable models` and `replaceable packages` and especially in these cases any user error usually leads to unrelated error messages. An automated system like this therefore makes such arrays much simpler to use.

4 RECURSIVE DATA STRUCTURES

Recursive data structures are among the oldest concepts in computer science [9, 13], yet they are not available in Modelica Standard 3.3 [1]. Only the generalized programming language MetaModelica, used in the implementation of OpenModelica, supports this concept [6, 7, 12]. Yet they can already be implemented in Modelica.

According to [6, Chapter 6.15.1] the omission of recursive data types was a conscious decision to avoid heap allocation. The implementation shown here doesn't require heap-allocation either for all data definitions are handeld during evaluation and are fixed after flattening.

Listing 4: Tree data global data array

```
1  model TreeData
2    replaceable TreeElemData values[groupTotal.total];
3    parameter String treeName = getInstanceName();
4    protected
5    UID.GroupTotal groupTotal(group=treeName);
6  end TreeData;
```

Listing 5: Tree data type element

```
1  record TreeElemData
2    Integer parent;
3    Integer left;
4    Integer right;
5    TreeElementType elemType;
6  end TreeElemData;
```

Listing 6: Tree data connectors

```
1  connector TreeConnector_a
2    input Integer up;
3    output Integer down;
4  end TreeConnector_a;
5
6  connector TreeConnector_b
7    output Integer up;
8    input Integer down;
9  end TreeConnector_b;
```

Listing 7: Tree data elements

```
1  model Root
2    TreeConnector_a con;
3    parameter String treeName = treeData.treeName;
4    protected
5    outer TreeData treeData;
6    UID.UniqueID uniqueID(group=treeName);
7    Integer ID = uniqueID.uid + 1;
8  equation
9    treeData.values[ID].parent = -1; // start of branch
10   treeData.values[ID].left = con.up;
11   treeData.values[ID].right = -2; // end of branch
12   con.down = ID;
13 end Root;
14
15 model Branch
16   TreeConnector_a con_left;
17   TreeConnector_a con_right;
18   TreeConnector_b con_parent;
19   parameter String treeName = treeData.treeName;
20   protected
21   outer TreeData treeData;
22   UID.UniqueID uniqueID(group=treeName);
23   Integer ID = uniqueID.uid+1;
24 equation
25   treeData.values[ID].parent = con_parent.down;
26   treeData.values[ID].left = con_left.up;
27   treeData.values[ID].right = con_right.up;
28   con_left.down = ID;
29   con_right.down = ID;
30   con_parent.up = ID;
31 end Branch;
32
33 model Leaf
34   TreeConnector_b con_parent;
35   parameter String treeName = treeData.treeName;
36   protected
37   outer TreeData treeData;
38   UID.UniqueID uniqueID(group=treeName);
39   Integer ID = uniqueID.uid+1;
40 equation
41   treeData.values[ID].parent = con_parent.down;
42   treeData.values[ID].left = -2;
43   treeData.values[ID].right = -2;
44   con_parent.up = ID;
45 end Leaf;
```

The idea used to implement those data structures is to use a global dynamic array of data elements (see Listing 4) and to use the position within this array as one would use a *pointer* in other languages, as demonstrated in Listing 5. In other words: we know we are operating on elements of type T in an array X. So `Integer` y can used as *pointer* to the element $T\ z = X[y]$. A value of ≤ 0 can't address any values and may therefore be used to indicate special conditions, like the end of a list, similar to other programming languages. Using special connectors in Listing 6 and the implementation outlined in Listing 7, a tree data structure can be constructed visually as shown in figure 1.

5 APPLICATIONS

The presented libraries find application in number of different simulations from a variety of different research fields.

Dynamic arrays are used in our multi-body contact simulations. For this purpose it has been slightly augmented to define a variable size matrix. In this matrix every row and every column corresponds to one uniquely identified object and a *Boolean* value indicates whether two objects are in contact (therefore the matrix is symmetric and the main axis defined to be always *false*). The presented library allows users to simply add new objects to the simulation. The new objects are then automatically added to the contact detection and the contact matrix grows automatically to accommodate the new value.

Recursive data structures have originally been created for the the DLR Rover Simulation Toolkit[8]. This toolkit allows users to quickly and easily create planetary rovers. When a new rover is assembled, internally a tree data structure is automatically created, which can then be parsed recursively. This is for example used for the rover controller to automatically calculate the rovers total mass.

6 PROPOSED NEW FEATURES FOR A FUTURE MODELICA VERSION

Unique IDs for C-Interfaces present a very specialized application: They are therefore well suited for a specialized library and need no inclusion into the Modelica standard. Compile-time dynamic arrays on the other hand are a very general concept with a large variety of possible applications and even so we have shown that they can in principle be implemented in Modelica, a simpler and standardized notation would be preferable. We propose that arrays of variable size use the same notation as is used in Modelica functions. Arrays without determined size are denoted by `A[:]` and their size real size can be determined using the notation `size(A, 1)`. This leaves only a notation to increase the size of an array. The proposed notation is `inc(A)`. This function may only be used in an *equation* section and not in a *when* or similar sub-section. The return value of the function would be a unique number in $\in [1, \texttt{size(1, A)}]$ (non-deterministic). An example using this simple notation can be found in Listing 8.

Listing 8: Proposal for dynamic array syntax

```
1  model DynamicArrayExample
2    Real A[:];
3    Integer i;
4    Integer j;
5  equation
6    i = inc(A);
7    A[i] = 5;
8    j = inc(A);
9    A[j] = 10;
10   // ⟹ A = {5, 10} or A = {10, 5}[3]
11   // ⟹ size(1, A) = 2
12 end DynamicArrayExample;
```

[3] depending on non-deterministic evaluation order but A[i] and A[j] are unambiguous

Just as dynamic arrays, recursive data structures can also be implemented, but a simpler and standardized notation would again be preferable. Modelica tries to abstract most implementation details and focuses on the user interface. The solution we presented is closely related to the concept of pointers, a very low level concept. For the inclusion into the Modelica standard we propose to move away from the presented implementation and to go for a more specialized approach. This would allow a more user-friendly treatment of the special cases discussed earlier. This part of the notation is critical, besides it is only necessary to use an *record* within itself[4].

The proposed syntax is shown in Listing 9. The *record* `test` contains a data value `a` and a recursive sub-element `rec` of type `test`. Access to `rec` is limited. The '='-operator is used to assign another record of type `test` to this element (similar to a *pointer*) and the function `getRec` to access the element (similar to dereferencing a *pointer*). The function returns the referenced *record* and a boolean indicating whether or not the dereferenciation was successful. If the referenced record is not defined as in line 12, the returned *record* is not defined and the second return value is `false` (line 13). After a value is assigned to the *record* in line 23, the value can be access in line 24. Of course this notation means that it may only be used in *algorithm* sections.

Listing 9: Proposal for recursive data structure syntax

```
1  record RecursiveRecord
2    Real a;
3    RecursiveRecord rec; // recursive definition
4  end RecursiveRecord;
5
6  function accessRecursiveRecordUnsuccessfull
7    RecursiveRecord t1;
8    RecursiveRecord t2;
9    Bool t2Success;
10 algorithm
11   t1.a = 5;
12   (t2, t2Success) = getRec(t1.rec);
13   // ⟹ t2Success = false; t2 = <undefined>
14 end accessRecursiveRecordUnsuccessfull;
15
16 function accessRecursiveRecordSuccessfull
17   RecursiveRecord t3;
18   RecursiveRecord t4;
19   RecursiveRecord t5;
20   Bool t5Success;
21 algorithm
22   t3.a = 7;
23   t4.rec = t3; // connect t4 with t3
24   (t5, t5Success) = getRec(t4.rec);
25   // ⟹ t5Success = true
26   // ⟹ t5.a = 7 (t5 is now alias for t3)
27 end accessRecursiveRecordSuccessfull;
```

Alternatively recursive data structures could be implemented in Modelica similar to MetaModelica. MetaModelica uses `uniontypes` to implement what they call `case records`. In short this construct allows a `records` member to be of variable type (with constraints). In the tree example every type shown in figure 1 would be of such a variable type. The branch would have two such recursive variable type elements for the left and right sub-branch, while leafs would have no such recursive elements [6, Chapter 6.15.1]. While this implementation is more flexible, we believe our proposal to be more user friendly.

[4]this might be more complicated when complex inheritance structures are considered but we will focus on the simpler case here

7 CONCLUSION

Dynamically sized arrays and recursive data structures are and have been at the core of almost all general purpose programming languages since their inception. For modeling languages, the integration of those concepts is more complicated. Changes to these data structures at run-time would necessitate structural changes to the underlying DAE system, a topic of ongoing research. But during compile-time such changes are possible. In fact Modelica already supports this for the specialized case of *expendable connectors* and in this paper we have demonstrated that it can be implemented in am more generalized manner for the Modelica IDE Dymola. But, for our implementation is both cumbersome and non-portable, we propose the inclusion of those concepts into the Modelica language.

When talking about dynamic data structures for modeling languages it is vital to differentiate between compile-time and run-time dynamic. While the later is hard and still requires a lot of research, the former can be implemented easily right now and can still significantly simplify the modeling process.

REFERENCES

[1] Modelica Association. 2014. *Modelica - A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.3 Revision 1.* PELAB, IDA, Linköpings Universitet, S-58183 Linköping, Sweden.
[2] Daniel Bender. 2016. DESA: Optimization of Variable Structure Modelica Models Using Custom Annotations. In *Proceedings of the 7th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT '16).* ACM, New York, NY, USA, 45–54.
[3] David Broman, Peter Fritzson, and Sébastien Furic. 2006. Types in the Modelica language. In *Proceedings of the 5th International Modelica Conference*, Dr. Christian Kral (Ed.). The Modelica Association, Vienna, Austria, 303–315.
[4] Dassault Systèmes. 2017. Multi-Engineering Modeling and Simulation - Dymola - CATIA. (2017). https://www.3ds.com/products-services/catia/products/dymola
[5] Hilding Elmqvist, Toivo Henningsson, and Martin Otter. 2016. *Systems Modeling and Programming in a Unified Environment Based on Julia.* Vol. 2. Springer International Publishing, Corfu, Greece, 198–217.
[6] Peter Fritzson. 2015. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach* (2 ed.). Wiley, Hoboken, NJ.
[7] P. Fritzson, A. Pop, and P. Aronsson. 2005. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, Prof. Dr.-Ing. Gerhard Schmitz (Ed.). The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology, Hamburg-Harburg, Germany, 519–525.
[8] Matthias Hellerer, Stefan Barthelmes, and Fabian Buse. 2017. The DLR Rover Simulation Toolkit. *14th Symposium on Advanced Space Technologies in Robotics and Automation.*
[9] IEEE Computer Society. 2000. Harold W. (Bud) Lawson - 2000 Computer Pioneer Award. (2000). https://www.computer.org/web/awards/pioneer-harold-lawson
[10] Donald E. Knuth. 1998. *The Art of Computer Programming* (3rd ed.). Fundamental Algorithms, Vol. 1. Addison Wesley Longman Publishing Co., Inc.
[11] National Institute of Standards and Technology. 2004. recursive data structures. (2004). https://xlinux.nist.gov/dads/HTML/recursivstrc.html
[12] Adrian Pop and Peter Fritzson. 2006. MetaModelica: A Unified Equation-based Semantical and Mathematical Modeling Language. In *Proceedings of the 7th Joint Conference on Modular Programming Languages (JMLC'06).* Springer-Verlag, Berlin, Heidelberg, 211–229.
[13] M. V. Wilkes. 1964. Lists and Why They Are Useful. In *Proceedings of the 1964 19th ACM National Conference (ACM '64).* ACM, New York, NY, USA, 61.1–61.5. https://doi.org/10.1145/800257.808911
[14] Dirk Zimmer. 2007. Enhancing Modelica towards variable structure systems. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (Linköping Electronic Conference Proceedings)*, Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen (Eds.). Linköping University Electronic Press; Linköpings universitet, Berlin, Germany, 61–70.
[15] Dirk Zimmer. 2010. *Equation-based modeling of variable-structure systems.* Ph.D. Dissertation. Eidgenössische Technische Hochschule (ETH), Zürich, Switzerland. Diss. ETH No. 18924.